Embedded System Tools Reference Manual

UG1043 (v2014.3) October 1, 2014

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/01/2014	2014.3	Vivado Design Suite release 2014.3. Removed information about Supervisory Call from Chapter 2, GNU Compiler Tools. Updated links to other documentation.
05/15/2014	2014.1	Vivado Design Suite release 2014.1. Initial release of document.

Table of Contents

Chapter 1

Embedded System and Tools Architecture Overview

This guide describes the architecture of the embedded system tools and flows provided in the Xilinx® Vivado® Design Suite for developing systems based on the MicroBlaze™ embedded processor and the Cortex A9 ARM processor.

The Vivado Design Suite system tools enable you to design a complete embedded processor system for implementation in a Xilinx FPGA device.

The Vivado Design Suite is a Xilinx development system product that is required to implement designs into Xilinx programmable logic devices. Vivado includes:

- The Vivado IP integrator tool, with which you can develop your embedded processor hardware.
- The Software Development Kit (SDK), based on the Eclipse open-source framework, which you can use to develop your embedded software application. SDK is also available as a standalone program.
- Embedded processing Intellectual Property (IP) cores including processors and peripherals.

For links to Vivado documentation and other useful information, see Appendix B, Additional Resources and Legal Notices.

Design Process Overview

The tools provided with Vivado are designed to assist in all phases of the embedded design process, as illustrated in Figure 1-1.

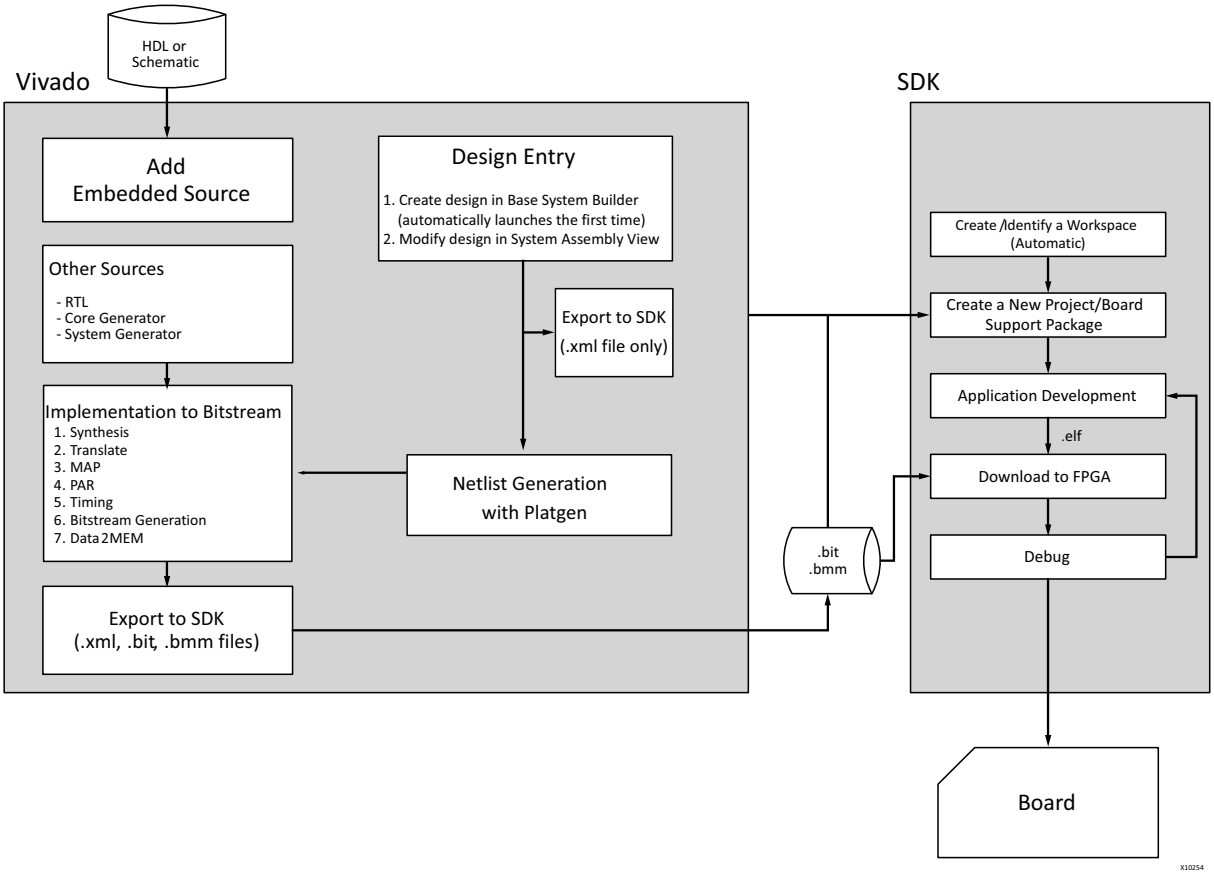

Figure 1-1: **Embedded Design Process Flow**

Hardware Development

Xilinx FPGA technology allows you to customize the hardware logic in your processor subsystem. Such customization is not possible using standard off-the-shelf microprocessor or controller chips.

The term "Hardware platform" describes the flexible, embedded processing subsystem you are creating with Xilinx technology for your application needs.

The hardware platform consists of one or more processors and peripherals connected to the processor buses.

When the hardware platform description is complete, the hardware platform can be exported for use by SDK.

Software Development

A board support package (BSP) is a collection of software drivers and, optionally, the operating system on which to build your application. The created software image contains only the portions of the Xilinx library you use in your embedded design. You can create multiple applications to run on the BSP.

The hardware platform must be imported into SDK prior to creation of software applications and BSP.

Verification

Vivado provides both hardware and software verification tools. The following subsections describe the verification tools available for hardware and software.

Hardware Verification Using Simulation

To verify the correct functionality of your hardware platform, you can create a simulation model and run it on an Hardware Design Language (HDL) simulator. When simulating your system, the processor(s) execute your software programs. You can choose to create a behavioral, structural, or timing-accurate simulation model.

Software Verification Using Debugging

The following options are available for software verification:

- You can load your design on a supported development board and use a debugging tool to control the target processor.
- You can gauge the performance of your system by profiling the execution of your code.

Device Configuration

When your hardware and software platforms are complete, you then create a configuration bitstream for the target FPGA device.

- For prototyping, download the bitstream along with any software you require to run on your embedded platform while connected to your host computer.
- For production, store your configuration bitstream and software in a non-volatile memory connected to the FPGA.

Vivado Design Suite Overview

An embedded hardware platform typically consists of one or more processors, peripherals and memory blocks, interconnected via processor buses. It also has port connections to the outside world. Each of the processor cores (also referred to as *pcores* or *processor IPs)* has a number of parameters that you can adjust to customize its behavior. These parameters also define the address map of your peripherals and memories. IP integrator lets you select from various optional features; consequently, the FPGA needs only implement the subset of functionality required by your application.

Figure 1-2 provides an overview of the Vivado architecture structure of how the tools operate together to create an embedded system.

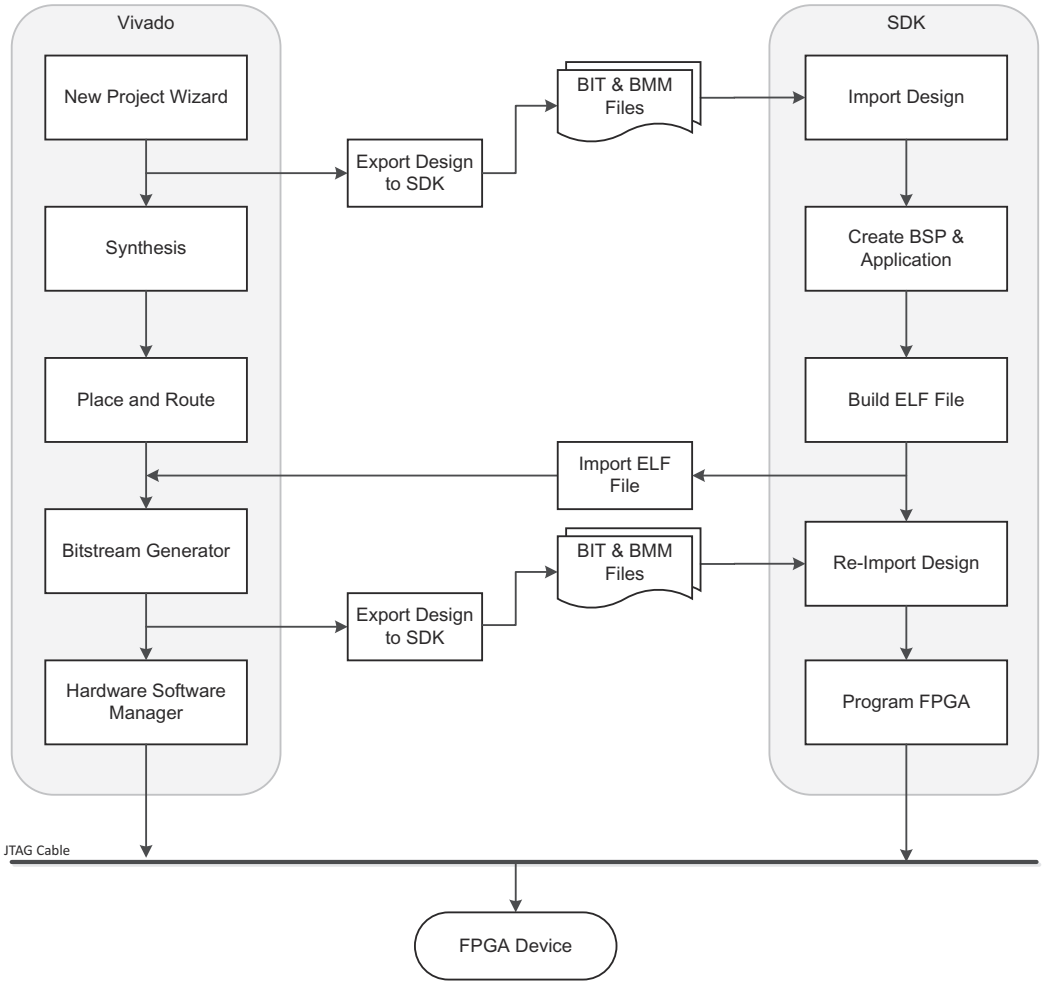

Figure 1-2: **Vivado Design Suite Tools Architecture**

Software Development Kit

The Software Development Kit (SDK) provides a development environment for software application projects. SDK is based on the Eclipse open-source standard. SDK has the following features:

- Can be installed independent of Vivado with a small disk footprint.
- Supports development of software applications on single- or multi-processor systems.
- Imports the Vivado-generated hardware platform definition.
- Supports development of software applications in a team environment.
- Ability to create and configure board support packages (BSPs) for third-party OS.
- Provides off-the-shelf sample software projects to test the hardware and software functionality.
- Has an easy GUI interface to generate linker scripts for software applications, program FPGA devices, and program parallel flash memory.
- Has feature-rich C/C++ code editor and compilation environment.
- Provides project management.
- Configures application builds and automates the make file generation.
- Supplies error navigation.
- Provides a well-integrated environment for seamless debugging and profiling of embedded targets.

For more information about SDK, see the *Software Development Kit (SDK) Help* (UG782) [Ref 1].

Table 1-1: **Software Development and Verification Tools**

GNU Compiler Tools	Builds a software application based on the platforms created.
Xilinx Microprocessor Debugger	Used for software download and debugging. Also provides a channel through which the GNU debugger accesses the device.
GNU Debugger	GUI for debugging software on either a simulation model or target device.
Flash Memory Programmer	Allows you to use your target processor to program on-board Common Flash Interface (CFI)-compliant parallel flash devices with software and data.

GNU Compiler Tools

GNU compiler tools (GCC) are called for compiling and linking application executables for each processor in the system. Processor-specific compilers are:

- The `mb-gcc` compiler for the MicroBlaze processor.
- The `arm-xilinx-eabi-gcc` compiler for the ARM processor.

As shown in the embedded tools architectural overview (Figure 1-2, page 8):

- The compiler reads a set of C-code source and header files or assembler source files for the targeted processor.
- The linker combines the compiled applications with selected libraries and produces the executable file in ELF format. The linker also reads a linker script, which is either the default linker script generated by the tools or one that you have provided.

Refer to Chapter 2, "GNU Compiler Tools,", Chapter 4, "GNU Debugger," and Appendix A, GNU Utilities for more information about GNU compiler tools and utilities.

Xilinx Microprocessor Debugger

You can debug your program in software using an Instruction Set Simulator (ISS), or on a board that has a Xilinx FPGA loaded with your hardware bitstream. As shown in Figure 1-2, page 8, the Xilinx Microprocessor Debugger (XMD) utility reads the application executable ELF file. For debugging on a physical FPGA, XMD communicates over the same download cable as used to configure the FPGA with a bitstream. Refer to Chapter 3, "Xilinx Microprocessor Debugger (XMD)," for more information.

GNU Debugger

The GNU Debugger (GDB) is a powerful yet flexible tool that provides a unified interface for debugging and verifying MicroBlaze and Cortex A9 ARM processor systems during various development phases.

GDB uses Xilinx Microprocessor Debugger (XMD) as the underlying engine to communicate to processor targets. Refer to Chapter 4, "GNU Debugger," for more information.

Flash Memory Programmer

The Flash Memory Programming solution is designed to be generic and targets a wide variety of flash hardware and layouts. See Chapter 5, "Flash Memory Programming."

GNU Compiler Tools

Overview

The Vivado® Design Suite includes the GNU compiler collection (GCC) for the MicroBlaze™ processor and the Cortex A9 processor.

- The Vivado GNU tools support both the C and C++ languages.
- The MicroBlaze GNU tools include `mb-gcc` and `mb-g++` compilers, `mb-as` assembler and `mb-ld` linker.
- The Cortex A9 ARM processor tools include `arm-xilinx-eabi-gcc` and `arm-xilinx-eabi-g++` compilers, `arm-xilinx-eabi-as` assembler, and `arm-xilinx-eabi-ld` linker.
- The toolchains also include the C, Math, GCC, and C++ standard libraries.

The compiler also uses the common binary utilities (referred to as binutils), such as an assembler, a linker, and object dump. The MicroBlaze and ARM compiler tools use the GNU binutils based on GNU version 2.16 of the sources. The concepts, options, usage, and exceptions to language and library support are described Appendix A, "GNU Utilities."

Compiler Framework

This section discusses the common features of the MicroBlaze and Cortex A9ARM processor compilers. Figure 2-1 displays the GNU tool flow.

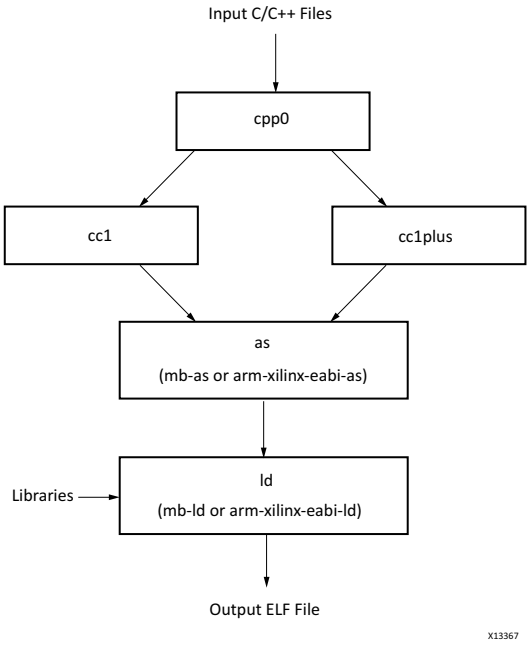

Figure 2-1: **GNU Tool Flow**

The GNU compiler is named `mb-gcc` for MicroBlaze and `arm-xilinx-eabi-gcc` for ARM Cores. The GNU compiler is a wrapper that calls the following executables:

- Pre-processor (`cpp0`)
  This is the first pass invoked by the compiler. The pre-processor replaces all macros with definitions as defined in the source and header files.
- Machine and language specific compiler
  This compiler works on the pre-processed code, which is the output of the first stage. The language-specific compiler is one of the following:
  - C Compiler (`cc1`)
    The compiler responsible for most of the optimizations done on the input C code and for generating assembly code.
  - C++ Compiler (`cc1plus`)
    The compiler responsible for most of the optimizations done on the input C++ code and for generating assembly code.
- Assembler (`mb-as` for MicroBlaze, `arm-xilinx-eabi-as` for ARM.
  The assembly code has mnemonics in assembly language. The assembler converts these to machine language. The assembler also resolves some of the labels generated by the compiler. It creates an object file, which is passed on to the linker.
- Linker (`mb-ld` for MicroBlaze, `arm-xilinx-eabi-ld for ARM`.
  Links all the object files generated by the assembler. If libraries are provided on the command line, the linker resolves some of the undefined references in the code by linking in some of the functions from the assembler.

Executable options are described in:

- Commonly Used Compiler Options: Quick Reference, page 17
- Linker Options, page 22
- MicroBlaze Compiler Options: Quick Reference, page 29
- MicroBlaze Linker Options, page 36
- ARM Cortex-A9 Compiler Usage and Options, page 45

Note: From this point forward the references to GCC in this chapter refer to the MicroBlaze compiler, `mb-gcc`, and references to G++ refer to the MicroBlaze C++ compiler, `mb-g++`.

Common Compiler Usage and Options

Usage

To use the GNU compiler, type:

```
<Compiler_Name> options files...
```

where <*Compiler_Name*> is `mb-gcc` or `arm-xilinx-eabi-gcc`. To compile C++ programs, you can use the `mb-g++` or `arm-xilinx-eabi-g++` command.

Input Files

The compilers take one or more of the following files as input:

- C source files
- C++ source files
- Assembly files
- Object files
- Linker scripts

Note: These files are optional. If they are not specified, the default linker script embedded in the linker (**`mb-ld`** or **`arm-xilinx-eabi-ld`**) is used.

The default extensions for each of these types are listed in Table 2-1. In addition to the files mentioned above, the compiler implicitly refers to the libraries files `libc.a`, `libgcc.a`, `libm.a`, and `libxil.a`. The default location for these files is the Vivado installation directory. When using the G++ compiler, the `libsupc++.a` and `libstdc++.a` files are also referenced. These are the C++ language support and C++ platform libraries, respectively.

Output Files

The compiler generates the following files as output:

- An ELF file. The default output file name is `a.exe` on Windows.
- Assembly file, if `-save-temps` or `-S` option is used.
- Object file, if `-save-temps` or `-c` option is used.
- Preprocessor output, `.i` or `.ii` file, if `-save-temps` option is used.

File Types and Extensions

The GNU compiler determines the type of your file from the file extension. Table 2-1 lists the valid extensions and the corresponding file types. The GCC wrapper calls the appropriate lower level tools by recognizing these file types.

Table 2-1: **File Extensions**

Extension	File type (Dialect)
`.c`	C file
`.C`	C++ file
`.cxx`	C++ file
`.cpp`	C++ file
`.c++`	C++ file
`.cc`	C++ file
`.S`	Assembly file, but might have preprocessor directives
`.s`	Assembly file with no preprocessor directives

Libraries

Table 2-2 lists the libraries necessary for the `mb_gcc` and `arm-xilinx-eabi-gcc` compilers.

Table 2-2: **Libraries Used by the Compilers**

Library	Particular
`libxil.a`	Contain drivers, software services (such as XilMFS) and initialization files developed for the Vivado tools.
`libc.a`	Standard C libraries, including functions like `strcmp` and `strlen`.
`libgcc.a`	GCC low-level library containing emulation routines for floating point and 64-bit arithmetic.
`libm.a`	Math Library, containing functions like `cos` and `sine`.
`libsupc++.a`	C++ support library with routines for exception handling, RTTI, and others.
`libstdc++.a`	C++ standard platform library. Contains standard language classes, such as those for stream I/O, file I/O, string manipulation, and others.

Libraries are linked in automatically by both compilers. If the standard libraries are overridden, the search path for these libraries must be given to the compiler. The `libxil.a` is modified to add driver and library routines.

Language Dialect

The GCC compiler recognizes both C and C++ dialects and generates code accordingly. By GCC convention, it is possible to use either the GCC or the G++ compilers equivalently on a source file. The compiler that you use and the extension of your source file determines the dialect used on the input and output files.

When using the GCC compiler, the dialect of a program is always determined by the file extension, as listed in Table 2-1, page 14. If a file extension shows that it is a C++ source file, the language is set to C++. This means that if you have compile C code contained in a CC file, even if you use the GCC compiler, it automatically mangles function names.

The primary difference between GCC and G++ is that G++ automatically sets the default language dialect to C++ (irrespective of the file extension), and if linking, automatically pulls in the C++ support libraries. This means that even if you compile C code in a `.c` file with the G++ compiler, it will mangle names.

Name mangling is a concept unique to C++ and other languages that support overloading of symbols. A function is said to be overloaded if the same function can perform different actions based on the arguments passed in, and can return different return values. To support this, C++ compilers encode the type of the function to be invoked in the function name, avoiding multiple definitions of a function with the same name.

Be careful about name mangling if you decide to follow a mixed compilation mode, with some source files containing C code and some others containing C++ code (or using GCC for compiling certain files and G++ for compiling others). To prevent name mangling of a C symbol, you can use the following construct in the symbol declaration.

```
#ifdef __cplusplus
extern "C" {
fendif

int foo();
int morefoo();

#ifdef __cplusplus
}
fendif
```

Make these declarations available in a header file and use them in all source files. This causes the compiler to use the C dialect when compiling definitions or references to these symbols.

Note: All Vivado drivers and libraries follow these conventions in all the header files they provide. You must include the necessary headers, as documented in each driver and library, when you compile with G++. This ensures that the compiler recognizes library symbols as belonging to "C" type.

When compiling with either variant of the compiler, to force a file to a particular dialect, use the `-x` *`lang`* switch. Refer to the GCC manual on the GNU website for more information on this switch. A link to the document is provided in the Appendix B, "Additional Resources and Legal Notices."

- When using the GCC compiler, `libstdc++.a` and `libsupc++.a` are *not* automatically linked in.
- When compiling C++ programs, use the G++ variant of the compiler to make sure all the required support libraries are linked in automatically.
- Adding `-lstdc++` and `-lsupc++` to the GCC command are also possible options.

For more information about how to invoke the compiler for different languages, refer to the GNU online documentation.

Commonly Used Compiler Options: Quick Reference

The summary below lists compiler options that are common to the compilers for MicroBlaze and ARM processors.

Note: The compiler options are case sensitive.

To jump to a detailed description for a given option, click its name in the table below.

General Options		Library Search Options
-E	-Wp,option	-l libraryname
-S	-Wa,option	-L Lib Directory
-c	-Wl,option	
-g	-help	**Header File Search Option**
-gstabs	-B directory	-I Directory Name
-On	-L directory	
-v	-I directory	**Linker Options**
-save-temps	-l library	-defsym _STACK_SIZE=value
-o filename		-defsym _HEAP_SIZE=value

General Options

-E

Preprocess only; do not compile, assemble and link. The preprocessed output displays on the standard out device.

-S

Compile only; do not assemble and link. Generates a `.s` file.

-c

Compile and Assemble only; do not link. Generates a `.o` file.

-g

This option adds DWARF2-based debugging information to the output file. The debugging information is required by the GNU debugger, `mb-gdb or arm-xilinx-eabi-gdb`. The debugger provides debugging at the source and the assembly level. This option adds debugging information only when the input is a C/C++ source file.

-gstabs

Use this option for adding STABS-based debugging information on assembly (`.s`) files and assembly file symbols at the source level. This is an assembler option that is provided directly to the GNU assembler, `mb-as` or `arm-xilinx-eabi-as`. If an assembly file is compiled using the compiler `mb-gcc` or `arm-xilinx-eabi-gcc`, prefix the option with **-Wa**.

-On

The GNU compiler provides optimizations at different levels. The optimization levels in the following table apply only to the C and C++ source files.

Table 2-3: **Optimizations for Values of n**

n	Optimization
0	No optimization.
1	Medium optimization.
2	Full optimization
3	Full optimization. Attempt automatic inlining of small subprograms.
S	Optimize for size.

Note: Optimization levels 1 and above cause code re-arrangement. While debugging your code, use of no optimization level is recommended. When an optimized program is debugged through **gdb**, the displayed results might seem inconsistent.

-v

This option executes the compiler and all the tools underneath the compiler in verbose mode. This option gives complete description of the options passed to all the tools. This description is helpful in discovering the default options for each tool.

-save-temps

The GNU compiler provides a mechanism to save the intermediate files generated during the compilation process. The compiler stores the following files:

- Preprocessor output -*input_file_name.i* for C code and *input_file_name.ii* for C++ code
- Compiler (`cc1`) output in assembly format - *input_file_name.s*
- Assembler output in ELF format - *input_file_name.s*

The compiler saves the default output of the entire compilation as `a.out`.

-o *filename*

The compiler stores the default output of the compilation process in an ELF file named `a.out`. You can change the default name using **-o** *output_file_name*. The output file is created in ELF format.

-Wp,*option*

-Wa,*option*

-Wl,*option*

The compiler, `mb-gcc` or `arm-xilinx-eabi-gcc`, is a wrapper around other executables such as the preprocessor, compiler (cc1), assembler, and the linker. You can run these components of the compiler individually or through the top level compiler.

There are certain options that are required by tools, but might not be necessary for the top-level compiler. To run these commands, use the options listed in the following table.

Table 2-4: **Tool-Specific Options Passed to the Top-Level GCC Compiler**

Option	Tool	Example
-Wp,*option*	Preprocessor	**mb-gcc** `-Wp,-D -Wp, MYDEFINE` ... Signal the pre-processor to define the symbol `MYDEFINE` with the **-D MYDEFINE** option.
-Wa,*option*	Assembler	`mb-as -Wa, ...` Signal the assembler to target the MicroBlaze processor.
-Wl,*option*	Linker	**mb-gcc** `-Wl,-M` ... Signal the linker to produce a map file with the **-M** option.

-help

Use this option with any GNU compiler to get more information about the available options. You can also consult the GCC manual.

-B directory

Add *directory* to the C run time library search paths.

-L directory

Add *directory* to library search path.

-I directory

Add *directory* to header search path.

-l library

Search *library* for undefined symbols.

Note: The compiler prefixes "lib" to the library name indicated in this command line switch.

Library Search Options

-l libraryname

By default, the compiler searches only the standard libraries, such as `libc`, `libm`, and `libxil`. You can also create your own libraries. You can specify the name of the library and where the compiler can find the definition of these functions. The compiler prefixes `lib` to the library name that you provide.

The compiler is sensitive to the order in which you provide options, particularly the **-l** command line switch. Provide this switch only after all of the sources in the command line.

For example, if you create your own library called `libproject.a`. you can include functions from this library using the following command:

```
Compiler Source_Files -L${LIBDIR} -l project
```

CAUTION! *If you supply the library flag* **-l** *library_name before the source files, the compiler does not find the functions called from any of the sources. This is because the compiler search is only done in one direction and it does not keep a list of available libraries.*

-L Lib Directory

This option indicates the directories in which to search for the libraries. The compiler has a default library search path, where it looks for the standard library. Using the **-L** option, you can include some additional directories in the compiler search path.

Header File Search Option

-I Directory Name

This option searches for header files in the */<dir_name>* directory before searching the header files in the standard path.

Default Search Paths

The compilers, `mb-gcc` and `arm-xilinx-eabi-gcc`, searches certain paths for libraries and header files. The search paths on the various platforms are described below.

Library Search Procedures

The compilers search libraries in the following order:

1. Directories are passed to the compiler with the **-L** <*dir_name*> option.
2. Directories are passed to the compiler with the **-B** <*dir_name*> option.
3. The compilers search the following libraries:

   a. `${XILINX_}/gnu/processor/platform/processor-lib/lib`

   b. `${XILINX_}/lib/processor`

   Note: `Processor` indicates `microblaze` for MicroBlaze, or `arm-xilinx-eabi` for ARM.

Header File Search Procedures

The compilers search header files in the following order:

1. Directories are passed to the compiler with the **-I** *<dir_name>* option.
2. The compilers search the following header files:

   a. `${XILINX_}/gnu/processor/platform/lib/gcc/processor/{gcc version}/include`

   b. `${XILINX_}/gnu/processor/platform/processor-lib/include`

Initialization File Search Procedures

The compilers search initialization files in the following order:

1. Directories are passed to the compiler with the **-B** *<dir_name>* option.
2. The compilers search `${XILINX_}/gnu/processor/platform/processor-lib/lib`.
3. The compilers search the following libraries:

   a. `$XILINX_/gnu/<processor>/platform/<processor-lib>/lib`

   b. `$XILINX_/lib/processor`

   Where:

   - *<processor>* is `microblaze` for MicroBlaze processors, and `arm-xilinx-eabi` for ARM processors
   - *<processor-lib>* is `microblaze-xilinx-elf` for MicroBlaze processors, and `arm-xilinx-eabi` for ARM processors.

   Note: *platform* indicates `lin` for Linux, `lin64` for Linux 64-bit and `nt` for Windows Cygwin.

Linker Options

-defsym _STACK_SIZE=value

The total memory allocated for the stack can be modified using this linker option. The variable _STACK_SIZE is the total space allocated for the stack. The _STACK_SIZE variable is given the default value of 100 words, or 400 bytes. If your program is expected to need more than 400 bytes for stack and heap combined, it is recommended that you increase the value of _STACK_SIZE using this option. The value is in bytes.

In certain cases, a program might need a bigger stack. If the stack size required by the program is greater than the stack size available, the program tries to write in other, incorrect, sections of the program, leading to incorrect execution of the code.

Note: A minimum stack size of 16 bytes (0x0010) is required for programs linked with the Xilinx-provided C runtime (CRT) files.

-defsym _HEAP_SIZE=value

The total memory allocated for the heap can be controlled by the value given to the variable _HEAP_SIZE. The default value of _HEAP_SIZE is zero.

Dynamic memory allocation routines use the heap. If your program uses the heap in this fashion, then you must provide a reasonable value for _HEAP_SIZE.

For advanced users: you can generate linker scripts directly from IP integrator.

Memory Layout

The MicroBlaze and ARM processors use 32-bit logical addresses and can address any memory in the system in the range 0x0 to 0xFFFFFFFF. This address range can be categorized into reserved memory and I/O memory.

Reserved Memory

Reserved memory has been defined by the hardware and software programming environment for privileged use. This is typically true for memory containing interrupt vector locations and operating system level routines. Table 2-5 lists the reserved memory locations for MicroBlaze and ARM processors as defined by the processor hardware. For more information on these memory locations, refer to the corresponding processor reference manuals.

For information about the ARM memory map, refer to the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 2].

Note: In addition to these memories that are reserved for hardware use, your software environment can reserve other memories. Refer to the manual of the particular software platform that you are using to find out if any memory locations are deemed reserved.

Table 2-5: **Hardware Reserved Memory Locations**

Processor Family	Reserved Memories	Reserved Purpose	Default Text Start Address
MicroBlaze	0x0 - 0x4F	Reset, Interrupt, Exception, and other reserved vector locations.	0x50
Cortex A9 ARM			

I/O Memory

I/O memory refers to addresses used by your program to communicate with memory-mapped peripherals on the processor buses. These addresses are defined as a part of your hardware platform specification.

User and Program Memory

User and Program memory refers to all the memory that is required for your compiled executable to run. By convention, this includes memories for storing instructions, read-only data, read-write data, program stack, and program heap. These sections can be stored in any addressable memory in your system. By default the compiler generates code and data starting from the address listed in Table 2-5 and occupying contiguous memory locations. This is the most common memory layout for programs. You can modify the starting location of your program by defining (in the linker) the symbol `_TEXT_START_ADDR` for MicroBlaze and `START_ADDR` for ARM.

In special cases, you might want to partition the various sections of your ELF file across different memories. This is done using the linker command language (refer to the Linker Scripts, page 27 for details). The following are some situations in which you might want to change the memory map of your executable:

- When partitioning large code segments across multiple smaller memories
- Remapping frequently executed sections to fast memories
- Mapping read-only segments to non-volatile flash memories

No restrictions apply to how you can partition your executable. The partitioning can be done at the output section level, or even at the individual function and data level. The resulting ELF can be non-contiguous, that is, there can be “holes” in the memory map. Ensure that you do not use documented reserved locations.

Alternatively, if you are an advanced user and want to modify the default binary data provided by the tools for the reserved memory locations, you can do so. In this case, you must replace the default startup files and the memory mappings provided by the linker.

Object-File Sections

An executable file is created by concatenating input sections from the object files (`.o` files) being linked together. The compiler, by default, creates code across standard and well-defined sections. Each section is named based on its associated meaning and purpose. The various standard sections of the object file are displayed in the following figure.

In addition to these sections, you can also create your own custom sections and assign them to memories of your choice.

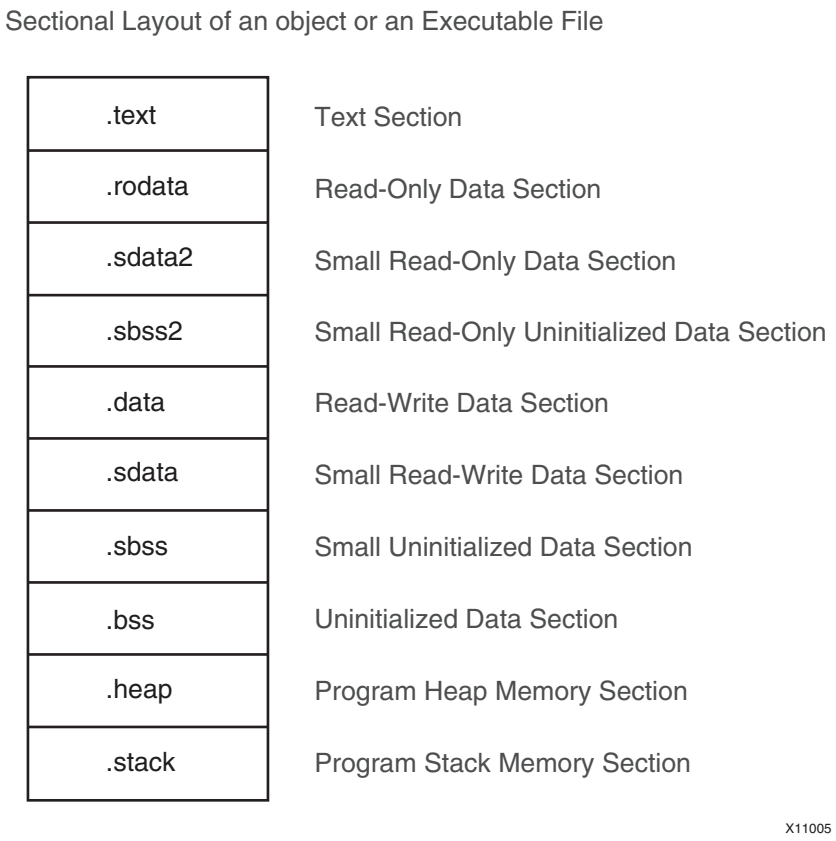

Figure 2-2: **Sectional Layout of an Object or Executable File**

The reserved sections that you would not typically modify include: `.init`, `.fini`, `.ctors`, `.dtors`, `.got`, `.got2`, and `.eh_frame`.

.text

This section of the object file contains executable program instructions. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags. This means that this section can be assigned to an initialized read-only memory (ROM) that is addressable from the processor instruction bus.

.rodata

This section contains read-only data. This section has the `r` (read-only) and the `i` (initialized) flags. Like the `.text` section, this section can also be assigned to an initialized, read-only memory that is addressable from the processor data bus.

.sdata2

This section is similar to the `.rodata` section. It contains small read-only data of size less than 8 bytes. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all the contents of this section are accessed using a single instruction. You can change the size of the data going into this section with the **-G** option to the compiler. This section has the `r` (read-only) and the `i` (initialized) flags.

.data

This section contains read-write data and has the `w` (read-write) and the `i` (initialized) flags. It must be mapped to initialized random access memory (RAM). It cannot be mapped to a ROM.

.sdata

This section contains small read-write data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all contents of the section can be accessed using a single instruction. This section has the `w` (read-write) and the `i` (initialized) flags and must be mapped to initialized RAM.

.sbss2

This section contains small, read-only un-initialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the r (read) flag and can be mapped to ROM.

.sbss

This section contains small un-initialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the `w` (read-write) flag and must be mapped to RAM.

.bss

This section contains un-initialized data. This section has the `w` (read-write) flag and must be mapped to RAM.

.heap

This section contains uninitialized data that is used as the global program heap. Dynamic memory allocation routines allocate memory from this section. This section must be mapped to RAM.

.stack

This section contains uninitialized data that is used as the program stack. This section must be mapped to RAM. This section is typically laid out right after the `.heap` section. In some versions of the linker, the `.stack` and `.heap` sections might appear merged together into a section named `.bss_stack`.

.init

This section contains language initialization code and has the same flags as `.text`. It must be mapped to initialized ROM.

.fini

This section contains language cleanup code and has the same flags as `.text`. It must be mapped to initialized ROM.

.ctors

This section contains a list of functions that must be invoked at program startup and the same flags as `.data` and must be mapped to initialized RAM.

.dtors

This section contains a list of functions that must be invoked at program end, the same flags as `.data`, and it must be mapped to initialized RAM.

.got2/.got

This section contains pointers to program data, the same flags as `.data`, and it must be mapped to initialized RAM.

.eh_frame

This section contains frame unwind information for exception handling. It contains the same flags as `.rodata`, and can be mapped to initialized ROM.

.tbss

This section holds uninitialized thread-local data that contribute to the program memory image. This section has the same flags as `.bss`, and it must be mapped to RAM.

.tdata

This section holds initialized thread-local data that contribute to the program memory image. This section must be mapped to initialized RAM.

.gcc_except_table

This section holds language specific data. This section must be mapped to initialized RAM.

.jcr

This section contains information necessary for registering compiled Java classes. The contents are compiler-specific and used by compiler initialization functions. This section must be mapped to initialized RAM.

.fixup

This section contains information necessary for doing fixup, such as the fixup page table, and the fixup record table. This section must be mapped to initialized RAM.

Linker Scripts

The linker utility uses commands specified in linker scripts to divide your program on different blocks of memories. It describes the mapping between all of the sections in all of the input object files to output sections in the executable file. The output sections are mapped to memories in the system. You do not need a linker script if you do not want to change the default contiguous assignment of program contents to memory. There is a default linker script provided with the linker that places section contents contiguously.

You can selectively modify only the starting address of your program by defining the linker symbol `_TEXT_START_ADDR` on MicroBlaze processors, or `START_ADDR` on ARM processors, as displayed in this example:

```
mb-gcc <input files and flags> -Wl,-defsym -Wl,_TEXT_START_ADDR=0x100
mb-ld <.o files> -defsym _TEXT_START_ADDR=0x100
```

The choices of the default script that will be used by the linker from the `$XILINX_/gnu/<procname>/<platform>/<processor_name>/lib/ldscripts` area are described as follows:

- `elf32<procname>.x` is used by default when none of the following cases apply.
- `elf32<procname>.xn` is used when the linker is invoked with the `-n` option.
- `elf32<procname>.xbn` is used when the linker is invoked with the `-N` option.
- `elf32<procname>.xr` is used when the linker is invoked with the `-r` option.
- `elf32<procname>.xu` is used when the linker is invoked with the `-Ur` option.

where <procname> =microblaze, <processor_name> = microblaze, and <platform> = lin or nt.

To use a linker script, provide it on the GCC command line. Use the command line option -T <script> for the compiler, as described below:

```
compiler -T <linker_script> <Other Options and Input Files>
```

If the linker is executed on its own, include the linker script as follows:

```
linker -T <linker_script> <Other Options and Input Files>
```

This tells GCC to use your linker script in the place of the default built-in linker script. Linker scripts can be generated for your program from within IP integrator and SDK.

In IP integrator or SDK, select **Tools > Generate Linker Script**.

This opens up the linker script generator utility. Mapping sections to memory is done here. Stack and Heap size can be set, as well as the memory mapping for Stack and Heap. When the linker script is generated, it is given as input to GCC automatically when the corresponding application is compiled within IP integrator or SDK.

Linker scripts can be used to assign specific variables or functions to specific memories. This is done through "section attributes" in the C code. Linker scripts can also be used to assign specific object files to sections in memory. These and other features of GNU linker scripts are explained in the GNU linker documentation, which is a part of the online binutils manual. A link to the GNU manuals is supplied in the Appendix B, "Additional Resources and Legal Notices." For a specific list of input sections that are assigned by MicroBlaze processor linker scripts, see "MicroBlaze Linker Script Sections" on page 37.

MicroBlaze Compiler Usage and Options

The MicroBlaze GNU compiler is derived from the standard GNU sources as the Xilinx port of the compiler. The features and options that are unique to the MicroBlaze compiler are described in the sections that follow. When compiling with the MicroBlaze compiler, the pre-processor provides the definition __MICROBLAZE__ automatically. You can use this definition in any conditional code.

MicroBlaze Compiler

The mb-gcc compiler for the Xilinx™ MicroBlaze soft processor introduces new options as well as modifications to certain options supported by the GNU compiler tools. The new and modified options are summarized in this chapter.

MicroBlaze Compiler Options: Quick Reference

Click an option name below to view its description.

Processor Feature Selection Options	General Program Options
-mcpu=vX.YY.Z	-msmall-divides
-mno-xl-soft-mul	-mxl-gp-opt
-mxl-multiply-high	-mno-clearbss
-mno-xl-multiply-high	-mxl-stack-check
-mxl-soft-mul	**Application Execution Modes**
-mno-xl-soft-div	-xl-mode-executable
-mxl-soft-div	-xl-mode-bootstrap
-mxl-barrel-shift	-xl-mode-novectors
-mno-xl-barrel-shift	**MicroBlaze Linker Options**
-mxl-pattern-compare	-defsym _TEXT_START_ADDR=value
-mno-xl-pattern-compare	-relax
-mhard-float	-N
-msoft-float	
-mxl-float-convert	
-mxl-float-sqrt	

Processor Feature Selection Options

-mcpu=v*X.YY.Z*

This option directs the compiler to generate code suited to MicroBlaze hardware version `v.X.YY.Z`. To get the most optimized and correct code for a given processor, use this switch with the hardware version of the processor.

The `-mcpu` switch behaves differently for different versions, as described below:

- `Pr-v3.00.a:` Uses 3-stage processor pipeline mode. Does not inhibit exception causing instructions being moved into delay slots.
- `v3.00.a and v4.00.a:` Uses 3-stage processor pipeline model. Inhibits exception causing instructions from being moved into delay slots.
- `v5.00.a` and later: Uses 5-stage processor pipeline model. Does not inhibit exception causing instructions from being moved into delay slots.

-mlittle-endian / -mbig-endian

Use these options to select the endianness of the target machine for which code is being compiled. The endianness of the binary object file produced is also set appropriately based on this switch. The GCC driver passes switches to the sub tools (`as`, `cc1`, `cc1plus`, `ld`) to set the corresponding endianness in the sub tool.

The default is `-mbig-endian`.

Note: You cannot link together object files of mixed endianness.

-mno-xl-soft-mul

This option permits use of hardware multiply instructions for 32-bit multiplications.

The MicroBlaze processor has an option to turn the use of hardware multiplier resources on or off. This option should be used when the hardware multiplier option is enabled on the MicroBlaze processor. Using the hardware multiplier can improve the performance of your application. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the usage of the multiplier option in MicroBlaze.

-mxl-multiply-high

The MicroBlaze processor has an option to enable instructions that can compute the higher 32-bits of a 32x32-bit multiplication. This option tells the compiler to use these multiply high instructions. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL_HIGH` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is available or not. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3],for more details about the usage of the multiply high instructions in MicroBlaze.

-mno-xl-multiply-high

Do not use multiply high instructions. This option is the default.

-mxl-soft-mul

This option tells the compiler that there is no hardware multiplier unit on MicroBlaze, so every 32-bit multiply operation is replaced by a call to the software emulation routine`__mulsi3`. This option is the default.

-mno-xl-soft-div

You can instantiate a hardware divide unit in MicroBlaze. When the divide unit is present, this option tells the compiler that hardware divide instructions can be used in the program being compiled.

This option can improve the performance of your program if it has a significant amount of division operations. The compiler automatically defines the C pre-processor definition `HAVE_HW_DIV` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the usage of the hardware divide option in MicroBlaze.

-mxl-soft-div

This option tells the compiler that there is no hardware divide unit on the target MicroBlaze hardware.

This option is the default. The compiler replaces all 32-bit divisions with a call to the corresponding software emulation routines (`__divsi3`, `__udivsi3`).

-mxl-barrel-shift

The MicroBlaze processor can be configured to be built with a barrel shifter. In order to use the barrel shift feature of the processor, use the option `-mxl-barrel-shift`.

The default option assumes that no barrel shifter is present, and the compiler uses add and multiply operations to shift the operands. Enabling barrel shifts can speed up your application significantly, especially while using a floating point library. The compiler automatically defines the C pre-processor definition `HAVE_HW_BSHIFT` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether or not this feature is specified as available. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the barrel shifter option in MicroBlaze.

-mno-xl-barrel-shift

This option tells the compiler not to use hardware barrel shift instructions. This option is the default.

-mxl-pattern-compare

This option activates the use of pattern compare instructions in the compiler.

Using pattern compare instructions can speed up boolean operations in your program. Pattern compare operations also permit operating on word-length data as opposed to byte-length data on string manipulation routines such as `strcpy`, `strlen`, and `strcmp`. On a program heavily dependent on string manipulation routines, the speed increase obtained will be significant. The compiler automatically defines the C pre-processor definition `HAVE_HW_PCMP` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the pattern compare option in MicroBlaze.

-mno-xl-pattern-compare

This option tells the compiler not to use pattern compare instructions. This is the default.

-mhard-float

This option turns on the usage of single precision floating point instructions (`fadd`, `frsub`, `fmul`, and `fdiv`) in the compiler.

It also uses `fcmp.p` instructions, where `p` is a predicate condition such as `le`, `ge`, `lt`, `gt`, `eq`, `ne`. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware. The compiler automatically defines the C pre-processor definition `HAVE_HW_FPU` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the hardware floating point unit option in MicroBlaze.

-msoft-float

This option tells the compiler to use software emulation for floating point arithmetic. This option is the default.

-mxl-float-convert

This option turns on the usage of single precision floating point conversion instructions (`fint` and `flt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

Refer to the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the hardware floating point unit option in MicroBlaze.

-mxl-float-sqrt

This option turns on the usage of single precision floating point square root instructions (`fsqrt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

Refer to the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the hardware floating point unit option in the MicroBlaze processor.

General Program Options

-msmall-divides

This option generates code optimized for small divides when no hardware divider exists. For signed integer divisions where the numerator and denominator are between 0 and 15 inclusive, this switch provides very fast table-lookup-based divisions. This switch has no effect when the hardware divider is enabled.

-mxl-gp-opt

If your program contains addresses that have non-zero bits in the most significant half (top 16 bits), then load or store operations to that address require two instructions.

The MicroBlaze processor ABI offers two global small data areas that can each contain up to 64 Kbytes of data. Any memory location within these areas can be accessed using the small data area anchors and a 16-bit immediate value, needing only one instruction for a load or store to the small data area. This optimization can be turned on with the `-mxl-gp-opt` command line parameter. Variables of size less than a certain threshold value are stored in these areas and can be addressed with fewer instructions. The addresses are calculated during the linking stage.

CAUTION! *If this option is being used, it must be provided to both the compile and the link commands of the build process for your program. Using the switch inconsistently can lead to compile, link, or run-time errors.*

-mno-clearbss

This option is useful for compiling programs used in simulation.

According to the C language standard, uninitialized global variables are allocated in the `.bss` section and are guaranteed to have the value 0 when the program starts execution. Typically, this is achieved by the C startup files running a loop to fill the `.bss` section with zero when the program starts execution. Optimizing compilers also allocates global variables that are assigned zero in C code to the `.bss` section.

In a simulation environment, the above two language features can be unwanted overhead. Some simulators automatically zero the entire memory. Even in a normal environment, you can write C code that does not rely on global variables being zero initially. This switch is useful for these scenarios. It causes the C startup files to not initialize the `.bss` section with zeroes. It also internally forces the compiler to not allocate zero-initialized global variables in the `.bss` and instead move them to the `.data` section. This option might improve startup times for your application. Use this option with care and ensure either that you do not use code that relies on global variables being initialized to zero, or that your simulation platform performs the zeroing of memory.

-mxl-stack-check

With this option, you can check whether the stack overflows when the program runs.

The compiler inserts code in the prologue of the every function, comparing the stack pointer value with the available memory. If the stack pointer exceeds the available free memory, the program jumps to a the subroutine `_stack_overflow_exit`. This subroutine sets the value of the variable `_stack_overflow_error` to 1.

You can override the standard stack overflow handler by providing the function `_stack_overflow_exit` in the source code, which acts as the stack overflow handler.

Application Execution Modes

-xl-mode-executable

This is the default mode used for compiling programs with `mb-gcc`. This option need not be provided on the command line for `mb-gcc`. This uses the startup file `crt0.o`.

-xl-mode-bootstrap

This option is used for applications that are loaded using a bootloader. Typically, the bootloader resides in non-volatile memory mapped to the processor reset vector. If a normal executable is loaded by this bootloader, the application reset vector overwrites the reset vector of the bootloader. In such a scenario, on a processor reset, the bootloader does not execute first (it is typically required to do so) to reload this application and do other initialization as necessary.

To prevent this, you must compile the bootloaded application with this compiler flag. On a processor reset, control then reaches the bootloader instead of the application.

Using this switch on an application that is deployed in a scenario different from the one described above will not work. This mode uses `crt2.o` as a startup file.

-xl-mode-novectors

This option is used for applications that do not require any of the MicroBlaze vectors. This is typically used in standalone applications that do not use any of the processor's reset, interrupt, or exception features. Using this switch leads to smaller code size due to the elimination of the instructions for the vectors. This mode uses `crt3.o` as a startup file.

CAUTION! *Do not use more than one mode of execution on the command line. You will receive link errors due to multiple definition of symbols if you do so.*

Position Independent Code

The GNU compiler for MicroBlaze supports the `-fPIC` and `-fpic` switches. These switches enable Position Independent Code (PIC) generation in the compiler. This feature is used by the Linux operating system only for MicroBlaze to implement shared libraries and relocatable executables. The scheme uses a Global Offset Table (GOT) to relocate all data accesses in the generated code and a Procedure Linkage Table (PLT) for making function calls into shared libraries. This is the standard convention in GNU-based platforms for generating relocatable code and for dynamically linking against shared libraries.

MicroBlaze Application Binary Interface

The GNU compiler for MicroBlaze uses the Application Binary Interface (ABI) defined in the *MicroBlaze Processor Reference Guide* (UG081) [Ref 3]. Refer to the ABI documentation for register and stack usage conventions as well as a description of the standard memory model used by the compiler.

MicroBlaze Assembler

The `mb-as` assembler for the Xilinx MicroBlaze soft processor supports the same set of options supported by the standard GNU compiler tools. It also supports the same set of assembler directives supported by the standard GNU assembler.

The `mb-as` assembler supports all the opcodes in the MicroBlaze machine instruction set, with the exception of the `imm` instruction. The mb-as assembler generates `imm` instructions when large immediate values are used. The assembly language programmer is never required to write code with `imm` instructions. For more information on the MicroBlaze instruction set, refer to the *MicroBlaze Processor Reference Guide* (UG081) [Ref 3].

The `mb-as` assembler requires all MicroBlaze instructions with an immediate operand to be specified as a constant or a label. If the instruction requires a PC-relative operand, then the mb-as assembler computes it and includes an `imm` instruction if necessary.

For example, the Branch Immediate if Equal (`beqi`) instruction requires a PC-relative operand.

The assembly programmer should use this instruction as follows:

```
beqi r3, mytargetlabel
```

where `mytargetlabel` is the label of the target instruction. The `mb-as` assembler computes the immediate value of the instruction as `mytargetlabel - PC`.

If this immediate value is greater than 16 bits, the `mb-as` assembler automatically inserts an `imm` instruction. If the value of `mytargetlabel` is not known at the time of compilation, the `mb-as` assembler always inserts an `imm` instruction. Use the `relax` option of the linker remove any unnecessary `imm` instructions.

Similarly, if an instruction needs a large constant as an operand, the assembly language programmer should use the operand as is, without using an `imm` instruction. For example, the following code adds the constant 200,000 to the contents of register r3, and stores the results in register r4:

```
addi r4, r3, 200000
```

The `mb-as` assembler recognizes that this operand needs an `imm` instruction, and inserts one automatically.

In addition to the standard MicroBlaze instruction set, the `mb-as` assembler also supports some pseudo-op codes to ease the task of assembly programming. Table 2-6 lists the supported pseudo-opcodes.

Table 2-6: **Pseudo-Opcodes Supported by the GNU Assembler**

Pseudo Opcodes	Explanation
`nop`	No operation. Replaced by instruction: `or R0, R0, R0`
`la Rd, Ra, Imm`	Replaced by instruction: `addik Rd, Ra, imm; = Rd = Ra + Imm;`
`not Rd, Ra`	Replace by instruction: `xori Rd, Ra, -1`
`neg Rd, Ra`	Replace by instruction: `rsub Rd, Ra, R0`
`sub Rd, Ra, Rb`	Replace by instruction: `rsub Rd, Rb, Ra`

MicroBlaze Linker Options

The `mb-ld` linker for the MicroBlaze soft processor provides additional options to those supported by the GNU compiler tools. The options are summarized in this section.

-defsym _TEXT_START_ADDR=*value*

By default, the text section of the output code starts with the base address 0x28. This can be overridden by using the `-defsym _TEXT_START_ADDR` option. If this is supplied to mb-gcc compiler, the text section of the output code starts from the given value.

You do not have to use `-defsym _TEXT_START_ADDR` if you want to use the default start address set by the compiler.

This is a linker option and should be used when you invoke the linker separately. If the linker is being invoked as a part of the mb-gcc flow, you must use the following option:

```
-Wl,-defsym -Wl,_TEXT_START_ADDR=value
```

-relax

This is a linker option that removes all unwanted `imm` instructions generated by the assembler. The assembler generates an `imm` instruction for every instruction where the value of the immediate cannot be calculated during the assembler phase.

Most of these instructions do not need an `imm` instruction. These are removed by the linker when the `-relax` command line option is provided.

This option is required only when linker is invoked on its own. When linker is invoked through the `mb-gcc` compiler, this option is automatically provided to the linker.

-N

This option sets the text and data section as readable and writable. It also does not page-align the data segment. This option is required only for MicroBlaze programs. The top-level GCC compiler automatically includes this option, while invoking the linker, but if you intend to invoke the linker without using GCC, use this option.

For more details on this option, refer to the GNU manuals online.

The MicroBlaze linker uses linker scripts to assign sections to memory. These are listed in the following section.

MicroBlaze Linker Script Sections

Table 2-7 lists the input sections that are assigned by MicroBlaze linker scripts.

Table 2-7: **Section Names and Descriptions**

Section	Description
`.vectors.reset`	Reset vector code.
`.vectors.sw_exception`	Software exception vector code.
`.vectors.interrupt`	Hardware Interrupt vector code.
`.vectors.hw_exception`	Hardware exception vector code.
`.text`	Program instructions from code in functions and global assembly statements.
`.rodata`	Read-only variables.
`.sdata2`	Small read-only static and global variables with initial values.
`.data`	Static and global variables with initial values. Initialized to zero by the boot code.
`.sdata`	Small static and global variables with initial values.
.sbss2	Small read-only static and global variables without initial values. Initialized to zero by boot code.
`.sbss`	Small static and global variable without initial values. Initialized to zero by the boot code.
`.bss`	Static and global variables without initial values. Initialized to zero by the boot code.
`.heap`	Section of memory defined for the heap.
`.stack`	Section of memory defined for the stack.

Tips for Writing or Customizing Linker Scripts

Keep the following points in mind when writing or customizing your own linker script:

- Ensure that the different vector sections are assigned to the appropriate memories as defined by the MicroBlaze hardware.
- Allocate space in the `.bss` section for stack and heap. Set the `_stack` variable to the location after `_STACK_SIZE` locations of this area, and the `_heap_start` variable to the next location after the `_STACK_SIZE` location. Because the stack and heap need not be initialized for hardware as well as simulation, define the `_bss_end` variable after the `.bss` and `COMMON` definitions.

  Note: The `.bss` section boundary does not include either stack or heap.
- Ensure that the variables `_SDATA_START__` , `_SDATA_END__`, `SDATA2_START`, `_SDATA2_END__`, `_SBSS2_START__` , `_SBSS2_END__`, `_bss_start`, `_bss_end`, `_sbss_start`, and `_sbss_end` are defined to the beginning and end of the sections `sdata`, `sdata2`, `sbss2`, `bss`, and `sbss` respectively.
- ANSI C requires that all uninitialized memory be initialized to startup (not required for stack and heap). The standard CRT that is provided assumes a single `.bss` section that is initialized to zero. If there are multiple `.bss` sections, this CRT will not work. You should write your own CRT that initializes all the `.bss` sections.

Startup Files

The compiler includes pre-compiled startup and end files in the final link command when forming an executable. Startup files set up the language and the platform environment before your application code executes. Start up files typically do the following:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers. Refer to Table 2-8, page 39 for details.
- Clear the BSS memory regions to zero.
- Invoke language initialization functions, such as C++ constructors.
- Initialize the hardware sub-system. For example, if the program is to be profiled, initialize the profiling timers.
- Set up arguments for the main procedure and invoke it.

Similarly, end files are used to include code that must execute after your program ends. The following actions are typically performed by end files:

- Invoke language cleanup functions, such as C++ destructors.
- De-initialize the hardware sub-system. For example, if the program is being profiled, clean up the profiling sub-system.

Table 2-8 lists the register names, values, and descriptions in the C-Runtime files.

Table 2-8: **Register Initialization in C-Runtime Files**

Register	Value	Description
r1	_stack-16	The stack pointer register is initialized to point to the bottom of the stack area with an initial negative offset of 16 bytes. The 16 bytes can be used for passing in arguments.
r2	_SDA2_BASE	_SDA2_BASE_ is the read-only small data anchor address.
r13	_SDA_BASE_	_SDA_BASE is the read-write small data anchor address.
Other registers	Undefined	Other registers do not have defined values.

The following subsections describe the initialization files used for various application modes. This information is for advanced users who want to change or understand the startup code of their application.

For MicroBlaze, there are two distinct stages of C runtime initialization. The first stage is primarily responsible for setting up vectors, after which it invokes the second stage initialization. It also provides exit stubs based on the different application modes.

First Stage Initialization Files

crt0.o

This initialization file is used for programs which are to be executed in standalone mode, without the use of any bootloader or debugging stub. This CRT populates the reset, interrupt, exception, and hardware exception vectors and invokes the second stage startup routine _crtinit. On returning from _crtinit, it ends the program by infinitely looping in the _exit label.

crt1.o

This initialization file is used when the application is debugged in a software-intrusive manner. It populates all the vectors *except the breakpoint and reset vectors* and transfers control to the second-stage _crtinit startup routine.

crt2.o

This initialization file is used when the executable is loaded using a bootloader. It populates all the vectors *except the reset vector* and transfers control to the second-stage _crtinit startup routine. On returning from _crtinit, it ends the program by infinitely looping at the _exit label. Because the reset vector is not populated, on a processor reset, control is transferred to the bootloader, which can reload and restart the program.

crt3.o

This initialization file is employed when the executable does not use any vectors and wishes to reduce code size. It populates only the reset vector and transfers control to the second stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `_exit` label. Because the other vectors are not populated, the GNU linking mechanism does not pull in any of the interrupt and exception handling related routines, thus saving code space.

Second Stage Initialization Files

According to the C standard specification, all global and static variables must be initialized to 0. This is a common functionality required by all the CRTs above. Another routine, `_crtinit`, is invoked. The `_crtinit` routine initializes memory in the `.bss` section of the program. The `_crtinit` routine is also the wrapper that invokes the `main` procedure. Before invoking the main procedure, it may invoke other initialization functions. The `_crtinit` routine is supplied by the startup files described below.

crtinit.o

This default, second stage, C startup file performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes "constructor" functions (`_init`).
4. Sets up the arguments for `main` and invokes `main`.
5. Invokes "destructor" functions (`_fini`).
6. Invokes `_program_clean` and returns.

pgcrtinit.o

This second stage startup file is used during profiling, and performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes `_profile_init` to initialize the profiling library.
4. Invokes "constructor" functions (`_init`).
5. Sets up the arguments for `main` and invokes `main`.
6. Invokes "destructor" functions (`_fini`).
7. Invokes `_profile_clean` to cleanup the profiling library.
8. Invokes `_program_clean`, and then returns.

sim-crtinit.o

This second-stage startup file is used when the `-mno-clearbss` switch is used in the compiler, and performs the following steps:

1. Invokes `_program_init`.
2. Invokes "constructor" functions (`_init`).
3. Sets up the arguments for `main` and invokes `main`.
4. Invokes "destructor" functions (`_fini`).
5. Invokes `_program_clean`, and then returns.

sim-pgcrtinit.o

This second stage startup file is used during profiling in conjunction with the `-mno-clearbss` switch, and performs the following steps in order:

1. Invokes `_program_init`.
2. Invokes `_profile_init` to initialize the profiling library.
3. Invokes "constructor" functions (`_init`).
4. Sets up the arguments for and invokes `main`.
5. Invokes "destructor" functions (`_fini`).
6. Invokes `_profile_clean` to cleanup the profiling library.
7. Invokes `_program_clean`, and then returns.

Other files

The compiler also uses certain standard start and end files for C++ language support. These are `crti.o`, `crtbegin.o`, `crtend.o`, and `crtn.o`. These files are standard compiler files that provide the content for the `.init`, `.fini`, `.ctors`, and `.dtors` sections.

Modifying Startup Files

The initialization files are distributed in both pre-compiled and source form with Vivado. The pre-compiled object files are found in the compiler library directory. Sources for the initialization files for the MicroBlaze GNU compiler can be found in the `<XILINX_>/sw/lib/microblaze/src` directory, where `<XILINX_>` is the Vivado installation area.

To fulfill a custom startup file requirement, you can take the files from the source area and include them as a part of your application sources. Alternatively, you can assemble the files into `.o` files and place them in a common area. To refer to the newly created object files instead of the standard files, use the `-B directory -name` command-line option while invoking `mb-gcc`.

To prevent the default startup files from being used, use the `-nostartfiles` on the final compile line.

Note: The miscellaneous compiler standard CRT files, such as `crti.o`, and `crtbegin.o`, are not provided with source code. They are available in the installation to be used as is. You might need to bring them in on your final link command.

Reducing the Startup Code Size for C Programs

If your application has stringent requirements on code size for C programs, you might want to eliminate all sources of overhead. This section describes how to reduce the overhead of invoking the C++ constructor or destructor code in a C program that does not require that code. You might be able to save approximately 220 bytes of code space by making the following modifications:

1. Follow the instructions for creating a custom copy of the startup files from the installation area, as described in the preceding sections. Specifically, copy over the particular versions of `crtn.s` and `xcrtinit.s` that suit your application. For example, if your application is being bootstrapped and profiled, copy `crt2.s` and `pg-crtinit.s` from the installation area.
2. Modify `pg-crtinit.s` to remove the following lines:

   ```
   brlid  r15, __init
   /* Invoke language initialization functions */
   nop
   ```

   and

   ```
   brlid  r15, __fini
   /* Invoke language cleanup functions */
   nop
   ```

   This avoids referencing the extra code usually pulled in for constructor and destructor handling, reducing code size.
3. Compile these files into `.o` files and place them in a directory of your choice, or include them as a part of your application sources.
4. Add the `-nostartfiles` switch to the compiler. Add the `-B directory` switch if you have chosen to assemble the files in a particular folder.
5. Compile your application.

If your application is executing in a different mode, then you must pick the appropriate CRT files based on the description in Startup Files, page 38.

Compiler Libraries

The `mb-gcc` compiler requires the GNU C standard library and the GNU math library. Precompiled versions of these libraries are shipped with Vivado. The CPU driver for MicroBlaze copies over the correct version, based on the hardware configuration of MicroBlaze. To manually select the library version that you would like to use, look in the following folder:

`$XILINX_/gnu/microblaze/<platform>/microblaze-xilinx-elf/lib`

The filenames are encoded based on the compiler flags and configurations used to compile the library. For example, `libc_m_bs.a` is the C library compiled with hardware multiplier and barrel shifter enabled in the compiler.

Table 2-9 shows the current encodings used and the configuration of the library specified by the encodings.

Table 2-9: **Encoded Library Filenames on Compiler Flags**

Encoding	Description
`_bs`	Configured for barrel shifter.
`_m`	Configured for hardware multiplier.
`_p`	Configured for pattern comparator.

Of special interest are the math library files (`libm*.a`). The C standard requires the common math library functions (`sin()` and `cos()`, for example) to use double-precision floating point arithmetic. However, double-precision floating point arithmetic may not be able to make full use of the optional, single-precision floating point capabilities in available for MicroBlaze.

The `Newlib` math libraries have alternate versions that implement these math functions using single-precision arithmetic. These single-precision libraries might be able to make direct use of the MicroBlaze processor hardware Floating Point Unit (FPU) and could therefore perform better.

If you are sure that your application does not require standard precision, and you want to implement enhanced performance, you can manually change the version of the linked-in library.

By default, the CPU driver copies the double-precision version (`libm_*_fpd.a`) of the library into your IP integrator project.

To get the single precision version, you can create a custom CPU driver that copies the corresponding `libm_*_fps.a` library instead. Copy the corresponding `libm_*_fps.a` file into your processor library folder (such as `microblaze_0/lib`) as `libm.a`.

When you have copied the library that you want to use, rebuild your application software project.

Thread Safety

The MicroBlaze processor C and math libraries distributed with Vivado are not built to be used in a multi-threaded environment. Common C library functions such as `printf()`, `scanf()`, `malloc()`, and `free()` are *not* thread-safe and will cause unrecoverable errors in the system at run-time. Use appropriate mutual exclusion mechanisms when using the Vivado libraries in a multi-threaded environment.

Command Line Arguments

The MicroBlaze processor programs cannot take command-line arguments. The command line arguments `argc` and `argv` are initialized to `0` by the C runtime routines.

Interrupt Handlers

Interrupt handlers must be compiled in a different manner than normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers must save the volatile registers that are being used. Interrupt handlers should also store the value of the machine status register (RMSR) when an interrupt occurs.

interrupt_handler attribute

To distinguish an interrupt handler from a sub-routine, mb-gcc looks for an attribute (`interrupt_handler`) in the declaration of the code. This attribute is defined as follows:

```
void function_name () __attribute__ ((interrupt_handler));
```

Note: The attribute for the interrupt handler is to be given *only* in the prototype and *not* in the definition.

Interrupt handlers might also call other functions, which might use volatile registers. To maintain the correct values in the volatile registers, the interrupt handler saves all the volatiles, if the handler is a non-leaf function.

Note: Functions that have calls to other sub-routines are called *non-leaf* functions.

Interrupt handlers are defined in the Microprocessor Software Specification (MSS) files. These definitions automatically add the attributes to the interrupt handler functions.

The interrupt handler uses the instruction `rtid` for returning to the interrupted function.

save_volatiles attribute

The MicroBlaze compiler provides the attribute `save_volatiles`, which is similar to the `interrupt_handler` attribute, but returns using `rtsd` instead of `rtid`.

This attribute saves all the volatiles for non-leaf functions and only the used volatiles in the case of leaf functions.

```
void function_name () __attribute__((save_volatiles));
```

fast_interrupt

The MicroBlaze compiler provides the attribute `fast_interrupt`, which is similar to the interrupt_handler attribute. On fast interrupt, MicroBlaze jumps to the interrupt routine address instead jumping to the fixed address 0x10.

Unlike a normal interrupt, when the attribute `fast_interrupt` is used on a C function, MicroBlaze saves only minimal registers.

```
void function_name () __attribute__ ((fast_interrupt));
```

Table 2-10: **Use of Attributes**

Attributes	Functions
interrupt_handler	This attribute saves the machine status register and all the volatiles, in addition to the non-volatile registers. `rtid` returns from the interrupt handler. If the interrupt handler function is a leaf function, only those volatiles which are used by the function are saved.
save_volatiles	This attribute is similar to `interrupt_handler`, but it uses `rtsd` to return to the interrupted function, instead of `rtid`.
fast_interrupt	This attribute is similar to `interrupt_handler`, but it jumps directly to the interrupt routine address instead of jumping to the fixed address 0x10.

ARM Cortex-A9 Compiler Usage and Options

ARM targets can be complied using Sourcery CodeBench Lite for Xilinx EABI.

Sourcery CodeBench contains the complete GNU Toolchain including all of the following components:

- CodeSourcery Common Startup Code Sequence
- CodeSourcery Debug Sprite for ARM
- GNU Binary Utilities (Binutils)
- GNU C Compiler (GCC)
- GNU C++ Compiler (G++)
- GNU C++ Runtime Library (Libstdc++)
- GNU Debugger (GDB)
- Newlib C Library

Usage

Compiling

```
arm-xilinx-eabi-gcc -c file1.c -I<include_path> -o file1.o
arm-xilinx-eabi-gcc -c file2.c -I<include_path> -o file2.o
```

Linking

```
arm-xilinx-eabi-gcc -Wl,-T -Wl,lscript.ld -L<libxil.a path> -o "App.elf"  file1.o
file2.o -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
```

For descriptions of flags used in the commands above, refer to the compiler help, using any of the following commands:

- arm-xilinx-eabi-gcc --help
- arm-xilinx-eabi-gcc -v --help
- arm-xilinx-eabi-gcc --target-help

Compiler Options

Other GNU compiler options that can be applied using ARM-related flags can be found on GNU Website: http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html. These flags can be used in the steps above, as required.

All the ARM GCC compiler options are listed at the link above. However, actual support depends on the target in use (ARM Cortex A9 in this case) and on the compiler toolchain.

For example:

The Sourcery CodeBench Lite for Xilinx EABI does not support `-mhard-float (-mfloat-abi=hard)`. Only soft and `softfp` floating point options are supported.

For more information on the toolchain, refer to the documentation available in the SDK installation path:

```
<Xilinx_Vivado_Installation_Path>\SDK\<2014.1>\gnu\arm\nt\share\doc
```

Other Notes

C++ Code Size

The GCC toolchain combined with the latest open source C++ standard library (`libstdc++-v3`) might be found to generate large code and data fragments as compared to an equivalent C program. A significant portion of this overhead comes from code and data for exception handling and runtime type information. Some C++ applications do not require these features.

To remove the overhead and optimize for size, use the `-fno-exceptions` and/or the `-fno-rtti` switches. This is recommended only for advanced users who know the requirements of their application and understand these language features. Refer to the GCC manual for more specific information on available compiler options and their impact.

C++ programs might have more intensive dynamic memory requirements (stack and heap size) due to more complex language features and library routines.

Many of the C++ library routines can request memory to be allocated from the heap. Review your heap and stack size requirements for C++ programs to ensure that they are satisfied.

C++ Standard Library

The C++ standard defines the C++ standard library. A few of these platform features are unavailable on the default Xilinx Vivado software platform. For example, file I/O is supported in only a few well-defined `STDIN`/`STDOUT` streams. Similarly, locale functions, thread-safety, and other such features may not be supported.

Note: The C++ standard library is not built for a multi-threaded environment. Common C++ features such as `new` and `delete` are not thread-safe. Please use caution when using the C++ standard library in an operating system environment.

For more information on the GNU C++ standard library, refer to the documentation available on the GNU website.

Position Independent Code (Relocatable Code)

The MicroBlaze processor compilers support the `-fPIC` switch to generate position independent code.

While both these features are supported in the Xilinx compiler, they are not supported by the rest of the libraries and tools, because Vivado only provides a standalone platform. No loader or debugger can interpret relocatable code and perform the correct relocations at runtime. These independent code features are not supported by the Xilinx libraries, startup files, or other tools. Third-party OS vendors could use these features as a standard in their distribution and tools.

Other Switches and Features

Other switches and features might not be supported by the Xilinx Vivado compilers and/or platform, such as `-fprofile-arcs`. Some features might also be experimental in nature (as defined by open source GCC) and could produce incorrect code if used inappropriately. Refer to the GCC manual for more information on specific features.

Xilinx Microprocessor Debugger (XMD)

The Xilinx® Microprocessor Debugger (XMD) is a tool that facilitates debugging programs and verifying systems using the MicroBlaze™ processor or the Dual ARM Cortex-A9 MPCore processor. You can use it to debug programs on MicroBlaze or Cortex A9 processors running on a hardware board, cycle-accurate Instruction Set Simulator (ISS).

XMD provides a Tool Command Language (Tcl) interface. This interface can be used for command line control and debugging of the target as well as for running complex verification test scripts to test a complete system.

XMD supports GNU Debugger (GDB) remote TCP protocol to control debugging of a target. Some graphical debuggers use this interface for debugging, including the MicroBlaze GDB (`mb-gdb`) and the Cortex A9 GDB (`arm-xilinx-eabi-gdb`), along with the Software Development Kit (SDK), the Eclipse-based software tool. In either case, the debugger connects to XMD running on the same computer or on a remote computer on the network.

XMD reads Xilinx Microprocessor Project the (XMP) system file or system.xml file, whichever is available, to gather information about the hardware system on which the program is debugged. The information is used to perform memory range tests, determine MicroBlaze to Microprocessor Debug Module (MDM) connectivity for faster download speeds, and perform other system actions.

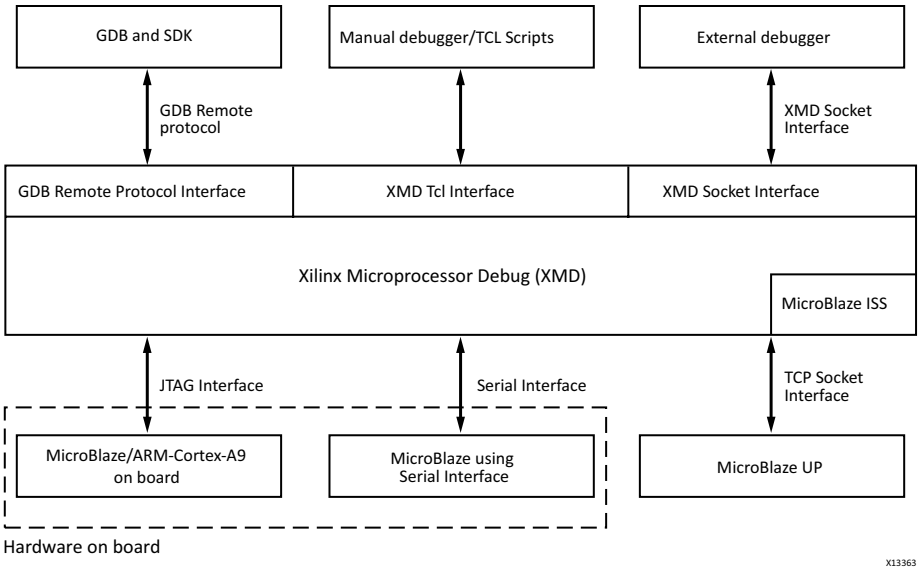

Figure 3-1: **XMD Targets**

XMD Usage

```
xmd [-h] [-help] [-hw <hardware_specification_file>] [-ipcport <port_number>][-nx]
[-opt <optfile>][-v] [-xmp <xmpfile>]
[-tcl <tcl_file> <tcl_args>]
```

XMD Options

Help

Displays the usage menu and quits.

Command: **-h, -help**

Hardware Specification File

Specifies the XML file that describes the hardware components.

Command: **-hw** *<hardware_specification_file>*

Port Number

Starts the XMD server at `<portnum>`. Internal XMD commands can be issued over this TCP Port. If `[<port_number>]` is not specified, a default value, `2345`, is used.

Command: **-ipcport** *<port_number>*

No Initialization File

Does not source `xmd.ini` file on startup.

Command: **-nx**

Option File

Specifies the option file to use to connect to target. The option file contains the XMD connect command to target.

Command: **-opt** *<connect_option_file>*

Tcl File

Specifies the XMD Tcl script to run.
The `<tclargs>` are arguments to the Tcl script. This Tcl file is sourced from XMD. XMD quits after executing the script. No other option can follow **-tcl**.
Command: **-tcl** *<tclfile> <tclarg>*

Version

Displays the version and then quits.

Command: **-v**

XMP File

Specifies the XMP file to load.

Command: **-xmp** <*xmpfile*>

Upon startup, XMD does the following:

- If an XMD Tcl script is specified, XMD executes the script, then quits.
- If an XMD Tcl script is *not* specified, XMD starts in *interactive mode*. In this case, XMD:
  a. Creates source `${HOME}/.xmdrc` file. You can use this configuration file to form custom Tcl commands using XMD commands:

- hw	loads the XML file. When -nx option is not given, sources the xmd.ini file if present in the current directory.
-opt	Uses Connect option file to connect to processor target.
-ipcport	opens XMD socket server
-xmp	loads system XMP file.

  b. Displays the `XMD%` prompt. From the `XMD Tcl` prompt, you can use XMD commands for debugging, as described in the next section, XMD Command Reference, page 52.

XMD Console

The XMD console is a standard Tcl console, where you can run any available Tcl commands. Additionally, the XMD console provides command editing convenience, such as file and command name auto-fill and command history.

The available Tcl commands on which you can use auto-fill are defined in the <*_Install_Area*>`/data/xmd/cmdlist` file. The command history is stored in `$HOME/.xmdcmdhistory`.

To use different files for available command names and command history, you can use environment variables `$XILINX_XMD_CMD_LIST` and `$XILINX_XMD_CMD_HISTORY` to overwrite the defaults.

XMD Command Reference

XMD User Command Summary

The following is a summary of XMD commands. To go to a description for a given command, click on its name in the list below.

User Commands		
bpl	dow -data	safemode [options]
bpr	elf_verify	srrd
bps	fpga -f <bitstream>	stackcheck
close_terminal	mrd <address> [<number of words|half words|bytes> {w|h|b}]	state
con		stats
connect	mrd_var	stop
cstp	mwr	stp
data_verify	profile	targets
debugconfig	read_uart	terminal
dis	rrd	tracestart
disconnect	rst	tracestop
dow	rwr	watch
	run	xload

XMD User Commands

bpl

Lists breakpoints and watchpoints.

Options	Example Usage
bpl	bpl

bpr

Removes breakpoints and watchpoints

Options	Example Usage
bpr {all|<bp id>|<address>|<function>}	bpr 0x400 bpr main bpr all

bps

Sets a software or hardware breakpoint at <*address*> or start of <*function name*>. The last downloaded ELF file is used for `function` lookup. Defaults to software breakpoint.

Options	Example Usage
`bps {<address>|<function_name>} {sw | hw}`	`bps 0x400` `bps main hw`

close_terminal

Closes the terminal server opened by the terminal command and the MDM Uart target connection.

Options	Example Usage
`close_terminal`	`close_terminal`

con

Continues from current PC or optionally specified *<Execute Start Address>*.

If `-block` option is specified, the command returns when the Processor stops on breakpoint or watchpoint.

A `-timeout` value can be specified to prevent indefinite blocking of the command.

The `-block` option is useful in scripting.

Options	Example Usage
`con [<Execute Start Address>] [-block [-timeout <Seconds>]]`	`con` `con 0x400`

connect

Connects to *<target_type>*. Valid target types are: `mb` and `mdm`. For additional information, refer to "Connect Command Options" on page 68.

Options	Example Usage
connect *<target_type(s)>*	**connect mb mdm**

cstp

Steps through the specified number of cycles. This is supported only on ISS targets.

Options	Example Usage
`cstp <number of cycles>`	`cstp` `cstp 10`

data_verify

Verify if the <*Binary filename*> is downloaded correctly to the target at <*Load Address*>.

Options	Example Usage
data_verify <*binary_filename*> <*load_address*>	**data_verify system.dat 0x400**

debugconfig

Configures the debug session for the target. For additional information, refer to "Configure Debug Session" on page 78.

Options	Example Usage
debugconfig	**debugconfig**
debugconfig -step_mode enable_interrupt	**debugconfig -step_mode** {disable_interrupt | enable_interrupt}
debugconfig -memory_datawidth_ matching enable	**debugconfig** -memory_datawidth_matching {disable | enable}
debugconfig -reset_on_run system enable	**debugconfig -reset_on_run** {system enable | processor enable | disable}
debugconfig -reset_on_data_dow processor enable	**debugconfig -reset_on_data_dow** {system enable | processor enable | disable}

dis

Disassemble instruction. Supported on the MicroBlaze target only.

Options	Example Usage
dis [<*address in hex*>] [<*number of words*>]	dis 0x400 10

disconnect

Disconnects from the current processor target, closes the corresponding GDB server, and reverts to the previous processor target, if any.

Options	Example Usage
disconnect <*target id*>	disconnect 0

dow

Downloads the given ELF or data file (with the -data option) onto the memory of the current target. If no address is provided along with the ELF file, the download address is determined from the ELF file by reading its headers.

Only those segments of the ELF file that are marked LOAD are written to memory.

Options	Example Usage
dow *<filename.elf>*	**dow executable.elf**
dow *<PIC filename.elf> <load_address>*	**dow executable.elf 0x400**

dow -data

If an address is provided with the ELF file (on MicroBlaze targets only), it is treated as Position Independent Code (PIC code) and downloaded at the specified address. Also, the R20 Register is set to the start address according to the PIC code semantics.

The R20 Register is reserved for storing a pointer to the Global Offset Table (GOT) in Position Independent Code (PIC). It is non-volatile in non-PIC code and must be saved across function calls.

When an ELF file is downloaded, the command does a reset, stops the processor at the reset location by using software breakpoints, and loads the ELF program to the memory. The reset is done to ensure that the system is in a known good state. The reset behavior can be configured using the following commands:

```
debugconfig -reset_on_run
{system enable | processor enable | disable}

debugconfig -reset_on_data_dow
{system enable | processor enable | disable}
```

Refer to the "Configure Debug Session" on page 78

Options	Example Usage
dow -data *<binary_filename> <load_address>*	**dow -data system.dat 0x400**

elf_verify

Verify if the executable.elf is downloaded correctly to the target. If ELF file is not specified, it uses the most recent ELF file downloaded on the target.

Options	Example Usage
elf_verify [*<filename.elf>*]	**elf_verify executable.elf**

fpga -f <bitstream>

Loads the FPGA device bitstream. Optionally specify the cable, JTAG configuration, and debug device options.

For additional information, refer to "Connect Command Options" on page 68.

Options	Example Usage
`fpga -f <bitstream>`	`fpga -f download.bit`
`fpga -f <bitstream> [-cable <cable_options>]| [-configdevice <configuration_options>] | [-debugdevice <device_name> ]`	`fpga -f download.bit -cable type xilinx_parallel`

mrd <address> [<number of words|half words|bytes> {w|h|b}]

Reads <num> memory locations starting at address. Defaults to a word (w) read.

If <Global Variable Name> name is specified, reads memory corresponding to global variable in the previously downloaded ELF file.

Options	Example Usage
mrd *<address> [<number of words| half words|bytes> {w|h|b}]*	**mrd 0x400** **mrd 0x400 10** **mrd 0x400 10 h**
mrd *<Global Variable Name>*	

mrd_var

Reads memory corresponding to global variable in the `<filename.elf>` or in a previously downloaded ELF file.

Options	Example Usage
mrd_var *<Global Variable Name> <filename.elf>*	**mrd_var global_var1 executable.elf**

mwr

Writes to num memory locations starting at <*address*> or <*Global Variable Name*>. Defaults to a word (w) write

Options	Example Usage
mwr <*address*> <*values*> [<*number of words/half words/bytes*> {w|h|b}]	mwr 0x400 0x12345678 mwr 0x400 0x1234 1 h mwr 0x400 {0x12345678 0x87654321} 2
mwr <*Global Variable Name*> <*values*> [<*number of words/half words/bytes*> {w|h|b}]	

profile

Writes a Profile output file, which can be interpreted by mb-gprof (for MicroBlaze), or arm-xilinx-eabi-gprof (for Cortex A9) to generate profiling information.

Specify the profile configuration sampling frequency in Hz, histogram bin size, and memory address for collecting profile data.

Options	Example Usage
profile [**-o** <*GMON Output filename*>]	**profile -o gproff.out**

read_uart

The read_uart start command redirects the output from the mdm UART interface to an optionally specified TCL channel (TCL Channel ID).

The read_uart stop command stops redirection.

A TCL channel represents an open file or a socket connection. The TCL channel should be opened prior to using the read_uart command, using appropriate TCL commands.

Options	Example Usage
read_uart [{start | stop}] [<*TCL Channel ID*>]	**read_uart start** **read_uart stop** **read_uart start $channel_id**

rrd

Reads all registers or reads *<reg_num>* register.

Options	Example Usage
rrd [*<reg_num>*]	**rrd** **rrd r1 (or) rrd R1** **rrd 1**

rst

Resets the system.

If the `-processor` option is specified, the current processor target is reset.

If the processor is not in a "Running" state (use the **state** command), then the processor will be stopped at the processor reset location on reset.

Options	Example Usage
rst [-processor]	**rst** **rst - processor**

rst for Zynq-7000 AP SoC Devices

For Zynq®-7000 AP SoC devices, some additional options are available for `rst`.

Get the active processor to trigger a soft reset. The system must be in a good state.

You can set hardware breakpoints with the bps command before issuing a reset. This is useful for debugging code booting from Flash.

Options	Description	Example Usage
`rst [-processor]`	Resets the processor.	**rst - processor**
`[-slcr]`	Triggers SLCR reset.	**rst - slcr**
`[-debug_sys]`	Triggers Debug System Reset through DAP.	**rst - debug_sys**
`[-srst]`	Asserts the SRST pin on the JTAG cable to trigger soft reset.	**rst -srst**

rwr

Registers writes from a *<register_number>*, *<register_name>*, or *<hex_value>*.

Options	Example Usage
rwr *<register_number>* | *register_name>* |*<Hex_value>*	**rwr pc 0x400**

run

Runs program from the program start address. The command does a "reset", stops the processor at the reset location by using breakpoints, and loads the ELF program data sections to the memory. Loading the ELF program data sections ensures that the static variables are properly initialized and "reset" is done so the system is in a "known good" state. The "reset" behavior can be configured using the following commands:

```
debugconfig -reset_on_run
{system enable | processor enable | disable}

debugconfig -reset_on_data_dow
{system enable | processor enable | disable}
```

Refer to "Configure Debug Session" on page 78.

Options	Example Usage
run	run

safemode [options]

Enables, disables, configures, and specifies files to be read in safemode. The following safemode options are available.

Options	Description	Example Usage
safemode [**-config** *<mode> <exception_mask>*]	Changes the current safemode configuration.	**safemode -config** *<mode> <exception_mask>*
safemode [{*on*|*off*}]	Enables and disables safemode.	**safemode on** **safemode off**
safemode [**-config** *<exception_id> <exception_addr>*]	Changes exception handler ID and/or addresses.	**safemode -config** *<exception_id>*
safemode[**-info**]	Displays the safemode information.	**safemode -info**
safemode [**-elf** *<elf_file>*]	Specifies the ELF file to be debugged.	**safemode -elf** *<elf_file>*

srrd

Reads special purpose registers or reads <*reg_name*> register.

Options	Example Usage
srrd	**srrd**
srrd [*<register_name>*]	**srrd pc**

stackcheck

Gives the stack usage information of the program running on the current target. The most recent ELF file downloaded on the target is taken into account for stack check.

Options	Example Usage
stackcheck	stackcheck

state

When no target id is specified, the command displays the current state of all targets.

When a *<target_id>* is specified, state of that target is displayed.

When **-system** *<system_id>* is specified the current state of all the targets in the system is displayed.

Options	Example Usage
state	state
state [*<target_id>*]	state *<target_id>*
state -system *<system_id>*	state *-system <system_id>*

stats

Displays execution statistics for the ISS target. The *<filename>* is the trace output from trace collection.

Options	Example Usage
stats	**stats**
stats [**<filename>**]	**stats trace.txt**

stop

Stops the target. For MicroBlaze, if the program is stalled at memory or FSL access, it is stopped forcibly.

Options	Example Usage
stop	stop

stp

Steps through the specified number of instructions.

Options	Example Usage
stp	stp
stp *<number of instructions>*	**stp 10**

targets

Lists information about all current targets or changes the current target.

Options	Example Usage
targets	**targets**
targets *<target_id>*	**targets 0**
targets -system *<system_id>*	**targets -system 1**

terminal

JTAG-based hyperterminal to communicate with mdm UART interface. The UART interface should be enabled in the mdm.

If the -jtag_uart_server option is specified, a TCP server is opened at *<port_no>*. Use any hyperterminal utility to communicate with opb_mdm UART interface over TCP sockets.

The *<port_number>* default value is 4321.

The *<baudrate>* determines the rate at which the JTAG UART port reads the data. This option can have the values low, med, or high. The default setting is med.

Increasing the baud rate might affect other debug operations, because XMD is busy polling for data on the JTAG UART port.

Options	Example Usage
terminal	**terminal**
terminal [**-jtag_uart_server**] [*<port_number>*][*<baudrate>*]	**terminal -jtag_uart_server 4321 high**

tracestart

Starts collecting instruction and function trace information to *<filename>*.

Trace collection can be stopped and started any time the program runs.

<filename> is specified on first tracestart only.

<pc_trace_filename> defaults to isstrace.out.

<func_trace_filename> defaults to fntrace.out.

Note: This is supported on ISS targets only.

Options	Example Usage
tracestart	**tracestart**
tracestart [*<pc_trace_filename>*] [**-function_name** *<func_trace_filename>*]	**tracestart pctrace.txt** **tracestart pctrace.txt -function_name fntrace.txt**

tracestop

Stops collecting trace information. The `done` option signifies the end of tracing.

Note: This is supported on ISS targets only.

Options	Example Usage
tracestop	**tracestop**
tracestop [done]	**tracestop done**

watch

Sets a read or write watchpoint at `address`. If the value compares to `data`, stop the processor.
Address and Data can be specified in hex `0x` format or binary `0b` format.
Don't care values are specified using `X`.
Addresses can be of contiguous range only.
Default value of data is `0xXXXXXXXX`. That is, it matches any value.

Options	Example Usage
watch {r | w} <*address*> [<*data*>]	**watch r 0x400 0x1234** **watch r 0x40X 0x12X4** **watch r 0b01000000XXXX 0b00010010XXXX0100** **watch r 0x40X**

xload

Loads hardware specification XML file. XMD reads the XML file to gather instruction and data memory address maps of the processor. This information is used to verify the program and data downloaded to processor memory. IP integrator generates the hardware specification file during the Export to SDK process.

Options	Example Usage
xload hw <hw_spec_file>	**xload hw system.xml**

Special Purpose Register Names

MicroBlaze Special Purpose Register Names

The following special register names are valid for MicroBlaze processors:

pc	msr	ear	esr	zpr
fsr	btr	pvr0	pvr1	zpr
pvr2	pvr3	pvr4	pvr5	zpr
pvr6	pvr7	pvr8	pvr9	
pvr10	pvr11	edr	pid	

For additional information, descriptions, and usage of MicroBlaze special register names, refer to the "Special Purpose Registers" section of the "MicroBlaze Architecture" chapter in the *MicroBlaze Processor Reference Guide* (UG081) [Ref 3].

Note: When MicroBlaze is debugged in `XMDSTUB` mode, only PC and MSR registers are accessible.

Important Cortex A9 Registers

Cortex A9 has a set of coprocessor registers. The different groups are listed here.

Table 3-1: **Cortex A9 Register Names**

ctrl	dma	tcm
id	etc	vfp

The examples section of this chapter contains details about these registers. Refer to the ARM documentation for additional information and for the complete set of ARM registers.

XMD Reset Sequence

When the `rst` command is issued, XMD resets the processor or system to bring them back to known states. Following are the sequences of operation that `rst` does for MicroBlaze processors.

1. Set a hardware breakpoint at reset location (`0x0`).
2. Issue reset signal (system reset or processor reset).
   The processor starts running.
3. After processor is stopped at reset location, remove the breakpoint.

Recommended XMD Flows

The following are the recommended steps in XMD for debugging a program and debugging programs in a multi-processor environment, and running a program in a debug session.

Debugging a Program

To debug a program:

1. Connect to the processor.
2. Download the ELF file.
3. Set the required breakpoints and watchpoints.
4. Start the processor execution using the **`con`** command or step through the program using the **`stp`** command.
5. Use the **`state`** command to check the processor status.
6. Use **`stop`** command to stop the processor if needed.
7. When the processor is stopped, read and write registers and memory.
8. To re-run the program, use the **`run`** command.

Debugging Programs in a Multi-Processor Environment

For debugging programs in a multi-processor environment:

1. Connect to processor1.
2. Use the **`debugconfig`** command to configure the reset behavior, which depends on your system architecture. Refer to the "Configure Debug Session" on page 78.
3. Download the ELF file.
4. Set the required breakpoints and watchpoints.
5. Start the processor execution using the **`con`** command or step through the program using the **`stp`** command.
6. Connect to processor2.
7. Use the **`debugconfig`** command to configure the reset behavior, which depends on your system architecture. Refer to the "Configure Debug Session" on page 78.
8. Download the ELF file.
9. Set the required Breakpoints and Watchpoints.
10. Start the processor execution using the **`con`** command or step through the program using the **`stp`** command.

11. Use the **targets** command to list the targets in the system. Each target is associated with a <*target id*>; an asterisk (*) marks the active target.
12. Use **targets** <*target id*> to switch between targets.
13. Use the **state** command to check the processor status.
14. Use the **stop** command to stop the processor.
15. When the processor is stopped, read and write the registers and memory.
16. To re-run the program use the **run** command.

Running a Program in a Debug Session

1. Connect to the processor.
2. Download the ELF file.
3. Set the Breakpoint at the <exit> function.
4. Start the processor execution using the **con** command.
5. Use the **state** command to check the processor status.
6. Use the **stop** command to stop the processor.
7. When the processor is stopped, read and write the registers and memory.
8. To re-run the program use the **run** command.

Using Safemode for Automatic Exception Trapping

XMD allows you to trap exceptions in your program when errors occur. Such errors include the execution of illegal instructions and bus errors. Use the following steps:

1. Download the program.
2. Run the **safemode on** command.
3. Start the program with the **con** command.

The program stops when an exception occurs. This feature is more useful when working with the GUI debugger (either Insight GDB or SDK).

- When using SDK, check the **Enable Safemode** checkbox box in the Initialization tab before running the program.
- When using GDB, download the program and run the **safemode on** command in XMD console before running the program in GDB.

When the exception occurs the program stops and the GUI shows the line of code that triggered the exception.

Processor Default Exception Settings

Table 3-2 shows the factory default settings for MicroBlaze exception trapping settings:

Table 3-2: **MicroBlaze Exception Settings**

Exception_id	Trap	Exception_Name
0	Yes	Fast Simplex Link exception.
1	No	Unaligned data access exception.
2	Yes	Illegal op-code exception.
3	Yes	Instruction bus error exception.
4	Yes	Data bus error exception.
5	Yes	Divide by zero exception.
6	Yes	Floating point unit exception.
7	Yes	Privileged instruction exception.
8	Yes	Data storage exception.
9	Yes	Instruction storage exception.
10	Yes	Data TLB miss exception.
11	Yes	Instruction TLB miss exception.
12	Yes	Assertion failure.
13	Yes	Program exit.

Overwriting Exception Settings

There are two methods to overwrite the default exception settings:

1. Use the command **xmdconfig** `[-mb_trap_mask] [MASK]`

   This sets the mask for all targets in the current XMD session. To define your own default setting for all XMD sessions, you can write that command in the `.xmdrc` file which is located at your home directory.

2. Use the command **safemode -config mode** `[MASK]`

   This sets the mask for current target only. While debugging a program, this is a convenient way to change the trap settings.

   Note: The current target is destroyed when you disconnect from the target.

Viewing Safemode Settings

You can view the current safemode setting with the **`safemode -info`** command.

In safe mode, XMD sets the breakpoint at the exception handlers that you want to trap.

For MicroBlaze processors, all exceptions take `PC` to `0x20`.

The detection works on most Standalone or Xilkernel projects If another software platform is used, the detection might fail. In such cases, set the exception handler address with the **`safemode -config`** *<exception_id> <exception_handler_addr>* command.

XMD ARM-Specific Commands

Table 3-3: ARM-Specific Commands

Command	Description
`mrc <CPx> <op1> <CRn> <CRm> <op2>`	Read ARM CoProcessor `CPx` Register specified by the parameters `CRn`, and `CRm`, and the opcodes `op1` and `op2`. The order of these parameters and opcodes is the same as the generic ARM MRC instruction.
`dp_rrd <reg offset>`	Read Debug Port Register available on ARM Debug Access Port (DAP)
`ap_rrd <reg offset>`	Read Access Port Register available on ARM Debug Access Port (DAP)
`mcr <CPx> <op1> <CRn> <CRm> <op2> <word>`	Write to ARM CoProcessor CPx Register specified by the parameters `CRn`, and `CRm`, and the opcodes `op1` and `op2`. The order of these parameters and opcodes is the same as the generic ARM MRC instruction.
`dp_rwr <reg offset>`	Write to Debug Port Register available on ARM Debug Access Port (DAP)
`ap_rwr <reg offset>`	Write to Access Port Register available on ARM Debug Access Port (DAP)
`mrd_phys <address> [num] [w|h|b]`	ARM Physical Memory Read through AHB Access Port. Read 'num' bytes/half-words/words at the address specified by <address> default: 'w'ord
`[-ocm]`	Read from OCM at 0x0 (if DDR is not remapped to 0x0)
`dmrd <address>`	Read CoreSight Memory available on ARM Debug Interface
`mwr_phys <addr> <values> [<num> <w|h|b>]`	ARM Physical Memory Write through AHB Access Port Write 'num' bytes/half-words/words at the address specified by <address> default: 'w'ord
`[-ocm]`	Write to OCM at 0x0 (if DDR is not remapped to 0x0)
`dmwr <address>`	Write to CoreSight Memory available on ARM Debug Interface

Connect Command Options

XMD can debug programs on different targets (processor or peripheral.)

- When communicating with a target, XMD connects to the target and a unique target ID is assigned to each target after connection.
- When connecting to a processor, the gdb server starts, enabling communication with GDB or SDK.

Usage

```
connect {mb | mdm | arm} <Connection_Type> [Options]
```

Table 3-4: **Connect Command Options**

Option	Description
mb	Connects to MicroBlaze processor
mdm	Connects to MDM peripheral
arm	Connects to Cortex- A9 processor and to the Coresight.
<Connection_Type>	Connection method, target dependent
[*Options*]	Connection options

The following sections describe connect options for different targets.

MicroBlaze Processor Target

XMD can connect through JTAG to one or more MicroBlaze processors using the MDM peripheral. XMD can communicate with a ROM monitor such as XMDStub through a JTAG or serial interface. You can also debug programs using built-in, cycle-accurate MicroBlaze ISS. The following sections describe the options for these targets.

MicroBlaze MDM Hardware Target

Use the command **connect mb mdm** to connect to the MDM target and start the remote GDB server. The MDM target supports non-intrusive debugging using hardware breakpoints and hardware single-step, without the need for a ROM monitor.

Figure 3-2, page 69 illustrates the MicroBlaze MDM target.

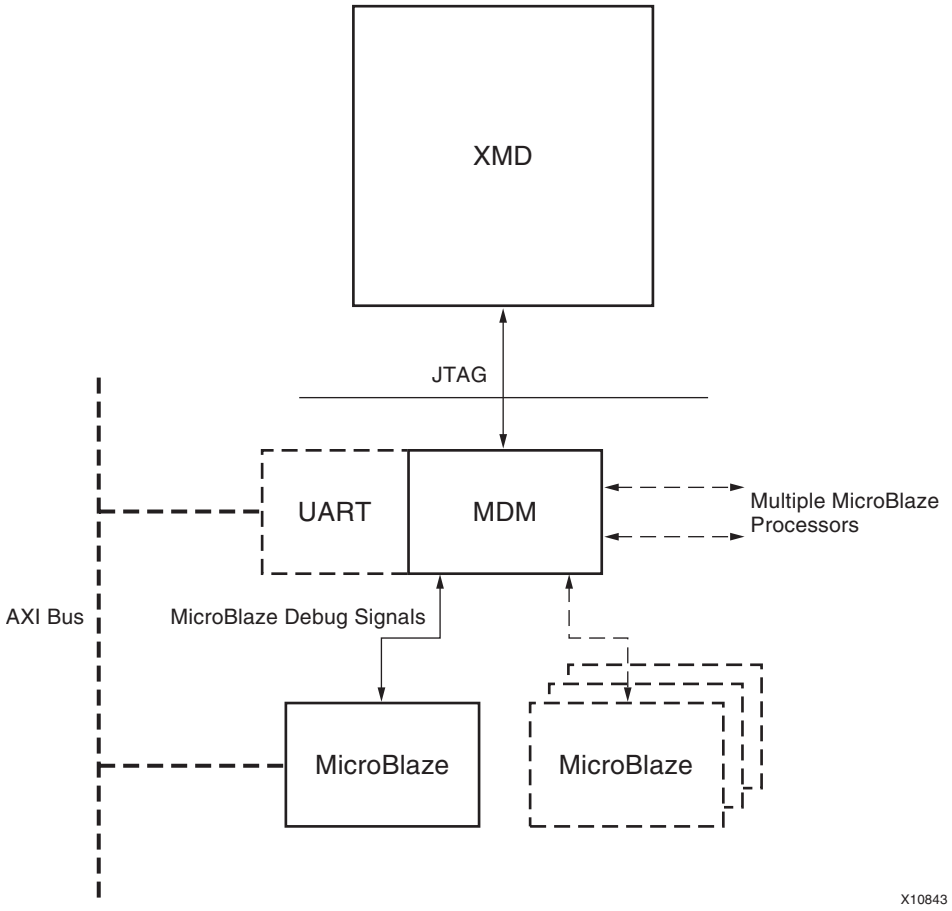

Figure 3-2: **MicroBlaze MDM Target**

When no option is specified to the **connect mb mdm**, XMD detects the JTAG cable automatically and chains the FPGA device containing the MicroBlaze-MDM system.

If XMD is unable to detect the JTAG chain or the FPGA device automatically, you can explicitly specify them using the following options:

Usage

```
connect mb hw [-cable <JTAG Cable options>] {[-configdevice <JTAG chain options>]}
[-debugdevice <MicroBlaze options>]
```

JTAG Cable Options and JTAG Chain Options

For JTAG cable and chain option descriptions, refer to Table 3-8, Program Initialization Option on page 85, and Table 3-9, Register/Memory Options on page 87, respectively.

MicroBlaze Options

Table 3-5 describes the MicroBlaze options.

Table 3-5: **MicroBlaze Options**

Option	Description
cpunr <CPU Number>	Specific MicroBlaze processor number to be debugged in an FPGA containing multiple MicroBlaze processors connected to MDM. The processor number starts from 1.
devicenr *<MicroBlaze device position>*	Position in the JTAG chain of the FPGA device containing the MicroBlaze processor. The device position number starts from 1.
romemstartadr <ROM start address>	Start address of Read-Only Memory. Use this to specify flash memory range. XMD sets hardware breakpoints instead of software breakpoints.
romemsize <ROM Size in Bytes>	Size of Read-Only Memory.
tlbstartadr <TLB start address>	Start address for reading and writing the Translation Look-aside Buffer (TLB).

MicroBlaze MDM Target Requirements

1. To use the hardware debug features on MicroBlaze, such as hardware breakpoints and hardware debug control functions like stopping and stepping, the hardware debug port must be connected to the MDM.
2. To use the UART functionality in the MDM target, you must set the C_USE_UART parameter while instantiating the MDM core in a system.

Note: Unlike the MicroBlaze stub target, programs should be compiled in executable mode and NOT in XMDSTUB mode while using the MDM target. Consequently, you do *not* need to specify an XMDSTUB_PERIPHERAL for compiling the XMDStub.

Example Debug Sessions

Example Using a MicroBlaze MDM Target

This example demonstrates a simple debug session with a MicroBlaze MDM target. Basic XMD-based commands are used after connecting to the MDM target using the `connect mb mdm` command. At the end of the session, `mb-gdb` connects to XMD using the GDB remote target. Refer to Chapter 4, "GNU Debugger," for more information about connecting GDB to XMD.

```
XMD% connect mb mdm
JTAG chain configuration
--------------------------------------------------
Device    ID Code         IR Length    Part Name
 1        0a001093           8
 2        f5059093          16         XCF32P
 3        01e58093          10         XC4VFX12
 4        49608093           8         xc95144xl

MicroBlaze Processor Configuration:
-------------------------------------
Version.............................7.00.a
Optimisation........................Performance
Interconnect........................PLBv46
No of PC Breakpoints................3
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Exceptions  Support.................off
FPU  Support........................off
Hard Divider Support................off
Hard Multiplier Support.............on - (Mul32)
Barrel Shifter Support..............off
MSR clr/set Instruction Support....on
Compare Instruction Support........on
PVR Supported.......................on
PVR Configuration Type..............Base

Connected to MDM UART Target
Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1234
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000510      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 00000140     r10: 00000000     r18: 00000000     r26: 00000000
    r3: a5a5a5a5     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 00000000     r13: 00000140     r21: 00000000     r29: 00000000
    r6: 00000000     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000064     r23: 00000000     r31: 00000000
    pc: 00000070     msr: 00000004
<--- Launching GDB from XMD% console --->
XMD% start mb-gdb microblaze_0/code/executable.elf
XMD%
<--- From GDB, a connection is made to XMD and debugging is done from the GDB GUI --->
XMD: Accepted a new GDB connection from 127.0.0.1 on port 3791
XMD%
```

```
XMD: GDB Closed connection
XMD% stp
BREAKPOINT at
     114:   F1440003  sbi     r10, r4, 3
XMD% dis 0x114 10
     114:   F1440003  sbi     r10, r4, 3
     118:   E0E30004  lbui    r7, r3, 4
     11C:   E1030005  lbui    r8, r3, 5
     120:   F0E40004  sbi     r7, r4, 4
     124:   F1040005  sbi     r8, r4, 5
     128:   B800FFCC  bri     -52
     12C:   B6110000  rtsd    r17, 0
     130:   80000000  Or      r0, r0, r0
     134:   B62E0000  rtid    r14, 0
     138:   80000000  Or      r0, r0, r0
XMD% dow microblaze_0/code/executable.elf
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING> stop
XMD% Info:User Interrupt, Processor Stopped at 0x0000010c
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING> rrd pc
pc : 0x000000f4 <--- With the MDM, the current PC of MicroBlaze can be        read
while the program is running
RUNNING> rrd pc
pc : 0x00000110 <--- Note: the PC is constantly changing, as the
                     program is running
RUNNING> stop
Info:Processor started. Type "stop" to stop processor
XMD% rrd
    r0: 00000000      r8: 00000065     r16: 00000000     r24: 00000000
    r1: 00000548      r9: 0000006c     r17: 00000000     r25: 00000000
    r2: 00000190     r10: 0000006c     r18: 00000000     r26: 00000000
    r3: 0000014c     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000500     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 24242424     r13: 00000190     r21: 00000000     r29: 00000000
    r6: 0000c204     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000068     r15: 0000005c     r23: 00000000     r31: 00000000
    pc: 0000010c     msr: 00000000
XMD% bps 0x100
Setting breakpoint at 0x00000100
XMD% bps 0x11c hw
Setting breakpoint at 0x0000011c
XMD% bpl
SW BP: addr = 0x00000100, instr = 0xe1230002 <-- Software Breakpoint
HW BP: BP_ID   0 : addr = 0x0000011c      <--- Hardware Breakpoint
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING>
Processor stopped at PC: 0x00000100
Info:Processor stopped. Type "start" to start processor
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING>
Info:Processor started. Type "stop" to stop processor
```

Example Using a Cortex A9 ARM Target

This example demonstrates a simple debug session with a Cortex A9 ARM target.

```
XMD% connect arm hwCortexA9
Processor Configuration
-----------------------------------
Version.............................0x00000003
User ID.............................0x00000000
No of PC Breakpoints................6
No of Addr/Data Watchpoints.........4
Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
XMD% rrd
r0: 0x00000000 r1: 0x00000000 r2: 0x00000000 r3: 0x00000000
r4: 0x00000000 r5: 0x00000000 r6: 0x00000000 r7: 0x00000000
r8: 0x00000000 r9: 0x00000000 r10: 0x00000000 r11: 0x00000000
r12: 0x00000000 sp: 0x00000000 lr: 0x00000000 pc: 0xffff0000
fpscr: 0x00120c20 cpsr: 0x000001d3
```

MicroBlaze Stub Hardware Target

To connect to a MicroBlaze target, use the `XMDStub` (a ROM monitor running on the target) and start a GDB server for the target. XMD connects to `XMDStub` through a JTAG or serial interface. The default option connects using a JTAG interface.

MicroBlaze Stub-JTAG Target Options

Usage

```
connect mb stub -comm jtag [-cable {<JTAG Cable options>}]
[-configdevice <{JTAG chain options>}] [-debugdevice {<MicroBlaze options>}]
```

MicroBlaze Option

Option	Description
devicenr <MicroBlaze device position>	The position in the JTAG chain of the FPGA device containing MicroBlaze.

MicroBlaze Stub-Serial Target Options

Usage

```
connect mb stub -comm serial {<Serial Communication options>}
```

Serial Communication Options

Table 3-6 lists the options that specify the MicroBlaze stub-serial target.

Table 3-6: **MicroBlaze Stub-Serial Target Options**

Option	Description
-baud *<serial port baud rate>*	Specifies the serial port baud rate in bits per second (bps). The default value is `19200` bps.
-port *<serial port>*	Specifies the serial port to which the remote hardware is connected when XMD communication is over the serial cable. The default serial ports are: ″ `/dev/ttyS0` on Linux ″ `Com1` on Windows
-timeout *<timeout in secs>*	Timeout period while waiting for a reply from XMDStub for XMD commands.

Note: If the program has any I/O functions such as `print() or putnum()` that write output onto the UART or MDM UART, it is printed on the console or terminal in which XMD was started.

Figure 3-3 illustrates a MicroBlaze sub target with an MDM UART and a UARTlite.

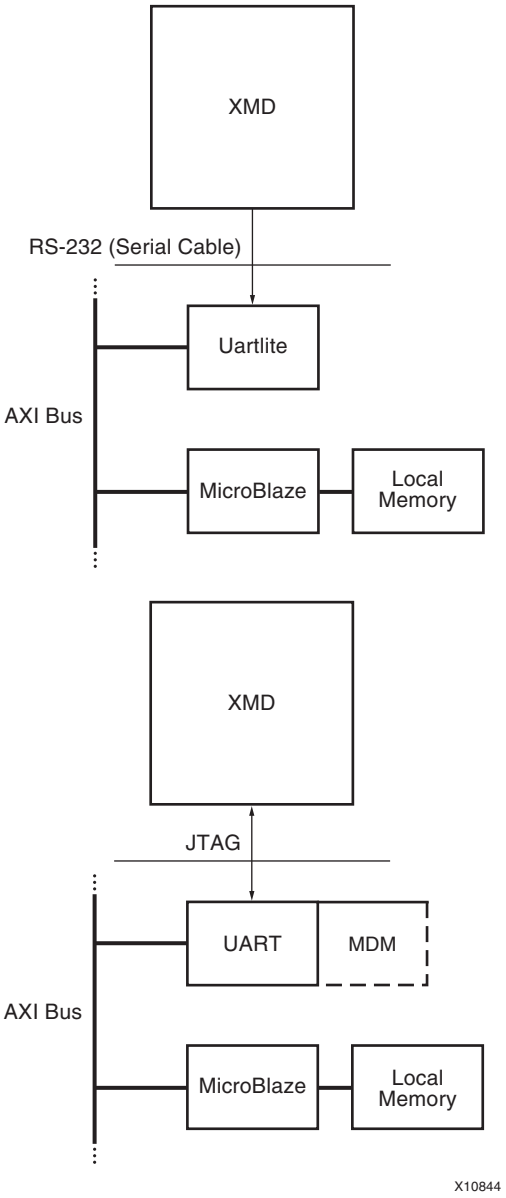

Figure 3-3: **MicroBlaze Stub Target with MDM UART and UARTlite**

Stub Target Requirements

To debug programs on the hardware board using XMD, the following requirements must be met:

- XMD uses a JTAG or serial connection to communicate with `XMDStub` on the board. Therefore, an mdm or a UART designated as `XMDSTUB_PERIPHERAL` in the MSS file is necessary on the target MicroBlaze system.

  Platform Generator can create a system that includes a mdm or a UART. The JTAG cables supported with the `XMDStub` mode are:

  - Xilinx parallel cable
  - Platform USB cable

- `XMDStub` on the board uses the MDM or UART to communicate with the host computer; therefore, it must be configured to use the MDM or UART in the MicroBlaze system.
- The `XMDStub` executable must be included in the MicroBlaze local memory at system startup.

  Data2MEM can populate the MicroBlaze memory with `XMDStub`. generates a Data2MEM script file that can be used to populate the block RAM contents of a bitstream containing a MicroBlaze system. It uses the executable specified in `DEFAULT_INIT`.

- For any program that must be downloaded on the board for debugging, the program start address must be higher than 0x800 and the program must be linked with the startup code in `crt1.o`.

  `mb-gcc` can compile programs satisfying the above two conditions when it is run with the option **`-xl-mode-xmdstub`**.

Note: For source level debugging, programs should also be compiled with the **`-g`** option. While initially verifying the functional correctness of a C program, it is advisable to not use any `mb-gcc` optimization option such as `-O2` or `-O3`, as `mb-gcc` performs aggressive code motion optimizations which might make debugging difficult to follow.

MicroBlaze Simulator Target

You can use `mb-gdb` and XMD to debug programs on the cycle-accurate simulator built in to XMD.

Usage

```
connect mb sim [-memsize <size>]
```

MicroBlaze Simulator Option

Option	Description
memsize *<size>*	The width of the memory address bus allocated in the simulator. Programs can access the memory range from **0** to **(2size)-1**. The default memory size is 64 KB.

Simulator Target Requirements

To debug programs on the Cycle-Accurate Instruction Set Simulator using XMD, you must compile programs for debugging and link them with the startup code in `crt0.o`.

The `mb-gcc` can compile programs with debugging information when it is run with the option `-g`, and by default, `mb-gcc` links `crt0.o` with all programs.

The option is **`-xl-mode-executable`**.

The program memory size must not exceed 64 K and must begin at address 0. The program must be stored in the first 64KB of memory.

Note: XMD with a simulator target does not support the simulation of OPB peripherals.

MDM Peripheral Target

You can connect to the `mdm` peripheral and use the UART interface for debugging and collecting information from the system.

Usage

```
connect mdm -uart
```

MDM Target Requirements

To use the UART functionality in the MDM target, you must set the `C_USE_UART` parameter while instantiating the mdm in a system.

UART input can also be provided from the host to the program running on MicroBlaze using the `xuart w <byte>` command. You can use the `terminal` command to open a hyperterminal-like interface to read and write from the UART interface. The `read_uart` command provides interface to write to STDIO or to file.

Configure Debug Session

Configure the debug session for a target using the **`debugconfig`** command. You can configure the behavior of instruction stepping and memory access method of the debugger.

Usage

```
debugconfig [-step_mode {disable_interrupt | enable_interrupt}]
[-memory_datawidth_matching {disable | enable}]
[-reset_on_run {system enable | processor enable | disable}]
[-reset_on_data_dow {system enable | processor enable | disable}]
```

Table 3-7 lists the debug configuration options.

Table 3-7: **Debug Configuration Options**

Option	Description
No Option	Lists the current debug configuration for the current session.
-step_mode {disable_interrupt | enable_interrupt}	Configures how XMD handles instruction stepping. disable_interrupt is the default mode. The interrupts are disabled during step. enable_interrupt enables interrupts during step. If an interrupt occurs during step, the interrupt is handled by the registered interrupt handler of the program.
-memory_datawidth_matching {disable | enable}	Configures how XMD handles memory read and write. By default, the data width matching is set to enable. All data width (byte, half, and word) accesses are handled using the appropriate data width access method. This method is especially useful for memory controllers and flash memory, where the datawidth access should be strictly followed. When data width matching is set to disable, XMD uses the best possible method, such as word access.
-reset_on_run [system enable | processor enable | disable]	Configures how XMD handles reset on program execution. A reset brings the system to a known consistent state for program execution. This ensures correct program execution without any side effects from a previous program run. By default, XMD performs system reset on run (on program download or program run). To enable different reset types, specify: **debugconfig -reset_on_run processor enable** **debugconfig -reset_on_run system enable** To disable reset, specify: **debugconfig -reset_on_run disable**
-reset_on_data_dow [system enable | processor enable | disable]	Changes how XMD handles reset on data download. A reset brings the system to a known consistent state for program execution. This ensures correct program execution without any side effects from a previous program run. By default, XMD performs system reset on run (on program download or program run). To enable different reset types, specify: **debugconfig -reset_on_data_dow processor enable** **debugconfig -reset_on_data_dow system enable** To disable reset, specify: **debugconfig -reset_on_data_dow disable**
-run_poll_interval *<time in millisec>*	When the processor is run using either the **run** or **con** command, XMD monitors the processor state at regular intervals (100 ms). If you want XMD to poll less frequently, use this option to specify the poll interval.

Configuring Instruction Step Modes

XMD supports two instruction step modes. You can use the `debugconfig` command to select between the modes. The two modes are:

- Instruction step with interrupts disabled:

  This is the default mode. In this mode the interrupts are disabled.

- Instruction step with interrupts enabled:

  In this mode the interrupts are enabled during step operation. XMD sets a hardware breakpoint at the next instruction and executes the processor.

  If an interrupt occurs, it is handled by the registered interrupt handler. The program stops at the next instruction.

Note: The instruction memory of the program should be connected to the processor d-side interface.

```
.XMD% debugconfig
Debug Configuration for Target 0
--------------------------------
Step Mode..................... Interrupt Disabled
Memory Data Width Matching... Disabled

XMD% debugconfig -step_mode enable_interrupt
XMD% debugconfig
Debug Configuration for Target 0
--------------------------------
Step Mode..................... Interrupt Enabled
Memory Data Width Matching... Disabled
```

Configuring Memory Access

XMD supports handling different memory data width accesses. The supported data widths are word (32 bits), half-word (16 bits), and Byte (8 bits). By default, XMD uses appropriate data width accesses when performing memory read and write operations. You can use the `debugconfig` command for configuring XMD to match the data width of the memory operation. This is usually necessary for accessing flash devices of different data widths.

```
XMD% debugconfig
Debug Configuration for Target 0
--------------------------------
Step Mode..................... Interrupt Disabled
Memory Data Width Matching... Enabled

XMD% debugconfig -memory_datawidth_matching disable
XMD% debugconfig
Debug Configuration for Target 0
--------------------------------
Step Mode..................... Interrupt Disabled
Memory Data Width Matching... Disabled
```

Configuring Reset for Multiprocessing Systems

By default, XMD performs a system reset upon download of a program to a processor. This behavior ensures a clean processor state before running the program. However, in multiprocessing systems, downloading and running programs to the various processors happens in sequence.

Depending upon the system architecture, a system reset performed during download of a program could cause programs downloaded to other processors, earlier in the sequence, to get reset. This may or may not be desirable; consequently, use the **`debugconfig`** command to disable system reset and or enable processor reset only on the various processors.

The following are example use cases:

Example 1: One Master Processor and Multiple Slave Processors

In this scenario, the program on the master processor gets downloaded and run first, followed by the other processors. In this case, the user wants to enable system reset on download to the master processor and only a processor reset (or no reset) on the other processors.

Example 2. Peer Processors

In this case, the download sequence could be arbitrary and the user wants to enable only processor reset (or no reset) at both the processors. This will ensure that downloading a program to one of the peer processors, does not affect the system state for the other peers.

Refer the `proc_sys_reset` IP module documentation for more information on how the reset connectivity and sequencing works through this module.

Cortex A9 Processor Target

```
XMD% connect arm hw

JTAG chain configuration
-------------------------------------------------
Device    ID Code         IR Length     Part Name
 1        4ba00477            4         Cortex-A9
 2        03727093            6         XC7Z020

CortexA9 Processor Configuration
-----------------------------------
Version.............................0x00000003
User ID.............................0x00000000
No of PC Breakpoints................6
No of Addr/Data Watchpoints.........1

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
```

`srrd [reg name]`	Special Register Read. For CortexA9, read a set of CoProcessor regs, identified by [reg name]. [reg name] can be any of ctrl, debug, dma, tcm, id, etc, vfp. (default: ctrl)
`mrc <CPx> <op1> <CRn> <CRm> <op2>`	Cortex A9 CoProcessor Register Read
`rwr <register> <word>`	Register Write
`mcr <CPx> <op1> <CRn> <CRm> <op2> <word>`	Cortex A9 CoProcessor Register Write
`mrd <address> [num] [w|h|b]`	Memory Read (default: 'w'ord)
`mrd_var <variable name> [ELF file]`	Read memory at global variable
`mrd_phys <address> [num] [w|h|b]`	Cortex A9 Memory Read through AHB AP (default: 'w'ord)
`[-force]`	Read from OCM at 0x0 (iff DDR is not remapped to 0x0)
`mwr <address> <values> [<num> <w|h|b>]`	Memory Write (default: 'w'ord)
`mwr_phys <address> <values> [<num> <w|h|b>]`	Cortex A9 Memory Write through AHB AP (default: 'w'ord)
`[-force]`	Write to OCM at 0x0 (iff DDR is not remapped to 0x0)

```
srrd ctrl
Control: 08c50879
Auxiliary Control: 00000045
Coprocessor Access Control: 00f00000
Secure Configuration: 00000000
Secure Debug Enable: 00000000
```

```
Non-Secure Access Control: 00000000
Translation Table Base 0: 00108059
Translation Table Base 1: 00108059
Translation Table Base Control: 00000002
Domain Access Control: ffffffff
Data Fault Status: 00000000
Instruction Fault Status: 00000000
Fault Address: 00000000
Watchpoint Fault Address: 00000000
Instruction Fault Address: 00000000
Secure or Non-secure Vector Base Address: 00100000
Monitor Vector Base Address: 00000000
Interrupt Status: 00000000
FCSE PID: 00000000
Context ID: : 00000000
User Read/Write Thread and Process ID: 00000000
User Read-only Thread and Process ID: 00000000
Privileged Only Thread and Process ID: 00000000
Peripheral Port Memory Remap: 00000000

XMD% srrd dbg
Unknown CortexA9 Register name dbg
XMD% srrd debug
Debug ID: 35137030
Debug Status and Control: 02086003
Data Transfer: f8000008
Watchpoint Fault Address: 00000000
Vector Catch: 00000000
Debug State Cache Control: 00000000
Debug State MMU Control: 00000000
Breakpoint Value 0: 00100000
Breakpoint Value 1: 00000000
Breakpoint Value 2: 00000000
Breakpoint Value 3: 00000000
Breakpoint Value 4: 00000000
Breakpoint Value 5: 00000000
Breakpoint Control 0: 004001e6
Breakpoint Control 1: 00000000
Breakpoint Control 2: 00000000
Breakpoint Control 3: 00000000
Breakpoint Control 4: 00000000
Breakpoint Control 5: 00000000
Watchpoint Value 0: 00000000
Watchpoint Value 1: 00000000
Watchpoint Control 0: 00000000
Watchpoint Control 1: 00000000

DMA Identification and Status present: 00100001
DMA Identification and Status queued: 00000000
DMA Identification and Status running: 00000000
DMA Identification and Status interrupting: 00000000
DMA User Accessibility: 00000000
DMA Channel Number: 00000000
DMA Control: 00000000
DMA Internal Start Address: 00000000
DMA External Start Address: 00000000
DMA Internal End Address: 00000000
DMA Channel Status: 00000000
DMA Context ID: 00000000
```

```
Data Cache Lockdown: 00000000
Instruction Cache Lockdown: 00000000
Data TCM Region: 00000000
Instruction TCM Region: 00000000
Data TCM Non-secure Control Access: 00000000
Instruction TCM Non-secure Control Access: 00000000
TCM Selection: 00000000
Cache Behavior Override: 00000000

Main ID: 413fc090
Cache Type: 83338003
TCM status: 00000000
TLB Type: 00000402
Processor Feature 0: 00001231
Processor Feature 1: 00000011
Debug Feature 0: 00010444
Auxiliary Feature 0: 00000000
Memory Model Feature 0: 00100103
Memory Model Feature 1: 20000000
Memory Model Feature 2: 01230000
Memory Model Feature 3: 00102111
Instruction Set Feature Attribute 0: 00101111
Instruction Set Feature Attribute 1: 13112111
Instruction Set Feature Attribute 2: 21232041
Instruction Set Feature Attribute 3: 11112131
Instruction Set Feature Attribute 4: 00011142
Instruction Set Feature Attribute 5: 00000000

XMD% srrd etc
PA: 00000000
Cache Dirty Status: 00000000
TLB Lockdown: 00000000
Primary Region Memory Remap: 00098aa4
Normal Region Memory Remap: 44e048e0
Secure User and Non-secure Access Validation Control: 00000000
Performance Monitor Control: 00000000
Cycle Counter: 00000000
Count 0: 00000000
Count 1: 00000000
Reset Counter: 00000000
Interrupt Counter: 00000000
Fast Interrupt Counter: 00000000
System Validation Cache Size Mask: 00000000
TLB Lockdown Index: 00000000
TLB Lockdown VA: 00000000
TLB Lockdown PA: 000000c6
TLB Lockdown Attributes: 00000000

XMD% srrd vfp
Floating-Point System ID: 41033094
Floating-Point Status And Control: 00000000
Floating-Point Exception: 40000000
Floating-Point Instruction: 40000000
Floating-Point Instruction 2: 40000000
Media and VFP Feature 0: 10110222
Media and VFP Feature 1: 01111111
```

XMD Internal Tcl Commands

In the Tcl interface mode, XMD starts a Tcl shell augmented with XMD commands. All XMD Tcl commands start with **x**, and you can list them from XMD by typing **x?**.

Xilinx recommends using the Tcl wrappers for these internal commands as described in XMD Options, page 50. The Tcl wrappers print the output of most of these commands and provide more options. While the Tcl wrappers are backward-compatible, the **x***<name>* commands will be deprecated.

The following Tcl command subsections are:

- Program Initialization Options
- Register/Memory Options
- Program Control Options
- Program Trace and Profile Options
- Miscellaneous Commands

Program Initialization Options

Table 3-8: **Program Initialization Option**

Option	Description
xconnect *<target>* {mb|mdm} *<connect type>* {*options*}	Connects to a processor or a peripheral target. Valid target types are mb and mdm. Refer to Connect Command Options, page 68 for more information on options.
xdebugconfig *<target id>* **[-step_mode** *<Step Type>***]** **[-memory_datawidth_matching {disable | enable}]** **[-reset_on_run** {system enable| processor enable | disable}] [**-reset_on_data_dow** {system enable | processor enable | disable}] [run_poll_interval *<time in millisec>*	Configures the debug session for the target. For additional information, refer to the Configure Debug Session, page 78.
xdisconnect [<*target id***>] [-cable]**	Disconnects from the target. Use the **-cable option** command to disconnect from cable and all targets.

Table 3-8: **Program Initialization Option *(Cont'd)***

Option	Description
xdownload *<target_id>* **<***filename***>** [*load address*] **xdownload** *<target_id>* **-data** *<filename> <load_address>*	Downloads the given ELF or data file, using the `-data` option, onto the memory of the current target. If no address is provided along with ELF file, the download address is determined from the ELF file headers. Otherwise, it is treated as Position Independent Code (PIC code) and downloaded at the specified address and Register R20 is set to the start address according to the PIC code semantics. XMD does not perform bounds checking, with the exception of preventing writes into the `XMDSTUB` area (address 0x0 to 0x800).
xrcableesn	Returns the ESN values of USB cables connected to the host machine.
xrjtagchain [-cable <cable_options>]	Returns the Jtag Device Chain information of the board connected to the host machine.
xfpga -f *<bitstream>* [**-cable** *<cable_options>*]| [**-configdevice** *<configuration_options>***]** | [**-debugdevice** *<device_name>*]	Loads the FPGA device bitstream and, optionally, the cable configuration and debug device options.
xload_sysfile hw *<hw_spec_file>*	Loads the hardware specification file.
xrut [*Session ID*]	Authenticates the XMD session when communicating over XMD sockets interface. The session ID is first assigned and subsequent calls return the session ID.
xtargets -listSysID **xtargets -system** <*system_ID*> [**-print**] [**-listTgtID**} **xtargets -target** *<target_ID>* {**-print** | **-prop**}	Provides system and target information in the current XMD session. `-listSysID` returns a list of existing systems. `-system` *<system_ID>* provides information on the specified system. `-print` prints the different targets in the system `-listTgtID` returns a list of existing targets in the system. `-target` *<target_ID>* provides information on the specified target. The options: `-print` prints the target information `-prop` returns the target properties

Register/Memory Options

Table 3-9: **Register/Memory Options**

Option	Description
xdata_verify <*target id*> <*Binary filename*><*load address*>	Verifies if the <*Binary filename*> was downloaded correctly at <*load address*> memory.
xdisassemble <*inst*>	Disassembles and displays one 32-bit instruction.
xelf_verify <*target id*> [<*filename*>.elf]	Verifies if the <*filename*>.elf is downloaded correctly to memory. If <*filename*>.elf is not specified, verifies the last downloaded ELF file to target.
xrmem <*target id*> <*address*> {<*number of bytes*|*half*|*word*>} {b | h | w} **xrmem** <*target id*> **-var** <*Global Variable Name*>	Reads <*number of bytes*> of memory locations from the specified memory address. Defaults to byte (**b**) read. Returns a list of data values. The data type depends on the data-width of memory access.
xwmem <*target id*> <*address*> {<*number of bytes*>|*half*|*word*} {b | h | w} <*value list*> **xwmem** <*target id*> **-var** <*Global Variable Name*> <*value list*>	Writes <*number of bytes*> data value from the specified memory address. Defaults to byte (**b**) write.
xrreg <*target id*> [reg]	Reads all registers or only register number <*reg*>.
xwreg <*target id*> [reg] [value]	Writes a 32-bit value into register number <*reg*>.
xstack_check <*target id*>	Gives the stack usage information of the program running on the current target. The most recent ELF file downloaded on the target is taken into account for stack check.

Program Control Options

Table 3-10: **Program Control Options**

Option	Description
xbreakpoint *<target id>* {*addr* | *function name*} {sw | hw}	Sets a breakpoint at the given address or start of function. **Note:** Breakpoints on instructions immediately following an IMM instruction can lead to undefined results for an `XMDStub` target.
xcontinue *<target id>* [*<Execute Start Address>*] [-block]	Continues from current PC or optionally specified *<Execute Start Address>*. If `-block` option is specified, the command returns when the Processor stops on breakpoint or watchpoint. The `-block` option is useful in scripting.
xcycle_step *<target id>* [cycles]	Cycle steps through one clock cycle. If `cycles` is specified, then step `cycles` number of clock cycles.[a]
xlist *<target id>*	Lists all of the breakpoint addresses.
xremove *<target id>* {*<addr>* | *<function name>* | *<bp id>* | all}	Removes one or more breakpoints or watchpoints.
xreset *<target id>* **[**reset type**]**	Resets target. Optionally, provide target-specific reset types such as the signals mentioned in Table 3-11 on page 89.
xrun *<target id>*	Runs program from the program start address.
xstate *<target id>*	Returns the processor target state; running or stopped.
xstep *<target id>*	Single steps one MicroBlaze instruction. If the PC is at an IMM instruction, the next instruction also runs. During a single step, interrupts are disabled by keeping the BIP flag set. Use **xcontinue** with breakpoints to enable interrupts while debugging.
xstop *<target id>*	Stops the program execution.
xwatch <target id> {r | w} <address> **[***<data value>***]**	Sets read/write watchpoints at a given *<address>* and, optionally, check for *<data value>*. If *<data value>* is not specified, watchpoints match any value. The address and value can be specified in hex or binary format.

a. This command is for Simulator targets only.

XMD MicroBlaze Hardware Target Signals

Table 3-11: **XMD MicroBlaze Hardware Target Signals**

Signal Name (Value)	Description
Non-maskable Break (0x10)	Similar to the Break signal, but works even while the BIP flag is already set. Refer the *MicroBlaze Processor Reference Guide* (UG081) [Ref 3] for more information about the BIP flag.
Processor Break (0x20)	Raises the Brk signal on MicroBlaze using the `JTAG UART Ext_Brk` signal. It sets the Break-in-Progress (BIP) flag on MicroBlaze and jumps to address 0x18.
Processor Reset (0x80)	Resets MicroBlaze using the `JTAG UART Debug_Rst` signal.
System Reset (0x40)	Resets the entire system by sending an OPB Rst using the `JTAG UART Debug_SYS_Rst` signal.

Program Trace and Profile Options

Table 3-12: **Program Trace/Profile Options**

Option	Description
xprofile <target id> [`-o <`*GMON Output File*`>`**]** **xprofile** <*target id*> `-config [sampling_freq_hw <value>] [binsize <value>] [profile_mem <start addr>]`	Generates profile output that can be read by `mb-gprof` (MicroBlaze) or `arm-xilinx-eabi-gprof` (Cortex A9). Specify the profile configuration sampling frequency in Hz, Histogram binary size, and memory address for collecting profile data.
xstats *<target id>* `{options}`	Displays the simulation statistics for the current session. Use the **reset** option to reset the simulation statistics.[a]
xtracestart *<target id>*	Starts collecting trace information.
xtracestop *<target id>*	Stops collecting trace information.[(a)]

a. This command is for ISS targets only.

Miscellaneous Commands

Table 3-13: **Miscellaneous Commands**

Command	Description
xclean	Cleans up any Xilinx resources that are using the cable.
xhelp	Lists the XMD commands.
xverbose	Toggles verbose mode on and off. Dumps debugging information from XMD.

GNU Debugger

This chapter describes the general usage of the Xilinx® GNU debugger (GDB) for the MicroBlaze™ and Cortex A9 processors.

Overview

GDB is a powerful and flexible tool that provides a unified interface for debugging and verifying MicroBlaze systems during various development phases. It uses Xilinx Microprocessor Debugger (XMD) as the underlying engine to communicate to processor targets.

Tool Overview

Tool Usage

MicroBlaze GDB usage:

```
mb-gdb <options> executable-file
```

ARM Coretx A9 GDB Usage:

```
arm-xilinx-eabi-gdb <options> executable
```

Tool Options

The following options are the most common in the GNU debugger:

`-command=FILE`	Execute GDB commands from the specified file. Used for debugging in batch and script mode.
`-batch`	Exit after processing options. Used for debugging in batch and script mode.
`-nx`	Do not read initialization file `.gdbinit`. If you have issues connecting to XMD (GDB connects and disconnects from XMD target), launch GDB with this option or remove the `.gdbinit` file.
`-nw`	Do not use a GUI interface.
`-w`	Use a GUI interface (Default).

Debug Flow Using GDB

1. Open Command prompt or terminal
2. In this session, run `settings64.bat` from `<Vivado_SDK_Installation_Path>\SDK\<2014.1>`
3. Type `XMD` to start XMD.
4. Connect to the Processor target. This action opens a GDB server for the target. For example:

```
XMD% connect arm hw
Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
```

5. For more XMD commands, refer to Chapter 3, Xilinx Microprocessor Debugger (XMD).
6. Open another command prompt session and repeat step 2.
7. In this session, start GDB by using a command suitable to the target in use (MicroBlaze or ARM Cortex A9 MP Core). For example:

```
arm-xilinx-eabi-gdb -nw executable.elf
```

8. In GDB, run `target remote <hostname>:<TCP port num>` to connect to the Remote GDB Server on XMD. For example, if 1234 is the TCP port no. from XMD, run:

```
<GDB> target remote localhost:1234
```

9. Download the Program and Debug application. In GDB, run the **load** command to download the program:

```
<GDB> load
```

MicroBlaze GDB Targets

The MicroBlaze GNU Debugger and XMD tools support remote targets. Remote debugging is done through XMD. The XMD server program can be started on a host computer with the Simulator target or the Hardware target.

The Cycle-Accurate Instruction Set Simulator (ISS) and the hardware interface provide powerful debugging tools for verifying a complete MicroBlaze system. The debugger `mb-gdb` connects to XMD using the GDB remote protocol over TCP/IP socket connection.

Simulator Target

The XMD simulator is a cycle-accurate ISS of the MicroBlaze system which presents the simulated MicroBlaze system state to GDB.

Hardware Target

With the hardware target, XMD communicates with Microprocessor Debug Module (`mdm`) debug core or an `xmdstub` program running on a hardware board through the serial cable or JTAG cable, and presents the running MicroBlaze system state to GDB.

For more information about XMD, see Chapter 3, Xilinx Microprocessor Debugger (XMD).

Compiling for Debugging on MicroBlaze Targets

To debug a program, you must generate debugging information when you compile the program. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code. The mb-gcc compiler for the Xilinx MicroBlaze soft processor includes this information when the appropriate modifier is specified.

The `-g` option in `mb-gcc` allows you to perform debugging at the source level. The debugger `mb-gcc` adds appropriate information to the executable file, which helps in debugging the code. The debugger mb-gdb provides debugging at source, assembly, and mixed source and assembly.

Note: While initially verifying the functional correctness of a C program, do not use any mb-gcc optimization option like `-O2` or `-O3` as `mb-gcc` does aggressive code motion optimizations which might make debugging difficult to follow.

Note: For debugging with XMD in hardware mode using `XMDSTUB`, specify the `mb-gcc` option `-xl-mode-xmdstub`. Refer to Chapter 3 for more information about compiling for specific targets.

ARM Cortex A9 Target

Debugging for the ARM Cortex A9 MP Cores is supported by `arm-xilinx-eabi-gdb` and XMD through the GDB Remote TCP protocol. Remote debugging is done through XMD.

To connect to an ARM Cortex A9 MP Core:

1. Start XMD and connect to the board using `connect arm hw` command, as described in Chapter 3, Xilinx Microprocessor Debugger (XMD).
2. Execute steps listed in Debug Flow Using GDB, page 91.

The debugger `arm-xilinx-eabi-gdb` attempts to make a connection to XMD. If successful, a message is printed in the shell window where XMD started. At this point, the debugger is connected to XMD and controls the debugging.

For GDB commands, refer to `refcard.pdf`, included in the SDK installation:

```
<Xilinx_Vivado_Installation_Path>\SDK\<2014.1>\gnu\arm\nt\share\doc\
xilinx-arm-xilinx-eabi\pdf
```

GDB Command Reference

For help on using `mb-gdb`, type `help` in the console mode.

To open a console window from the GBD main dialog box, select **View > Console**.

For comprehensive online documentation on using GDB, refer to the GNU web site. For information about the `mb-gdb` Insight GUI, refer to the Red Hat Insight website. Links to these documents are provided in Appendix B, Additional Resources and Legal Notices.

Table 4-1 describes the commonly used `mb-gdb` console commands. The equivalent GUI versions can be identified in the `mb-gdb` GUI window icons. Some of the commands, such as info target and monitor info, might be available only in the console mode.

Table 4-1: **Commonly Used GDB Console Commands**

Command	Description
`load <program>`	Load the program into the target.
`b main`	Set a breakpoint in function main.
`c`	Continue after a breakpoint. Do not use the `run` command
`l`	View a listing of the program at the current point.
`n`	Steps one line, stepping over function calls.
`s`	Step one line, stepping into function calls.
`stepi`	Step one assembly line.
`info reg`	View register values.
`info target`	View the number of instructions and cycles executed for the built-in simulator only.
`p <xyz>`	Print the value of *xyz* data.
`hbreak main`	Set hardware breakpoint in function `main()`.
`watch <gvar1>`	Set Watchpoint on Global Variable *gvar1*.
`rwatch <gvar1>`	Set Read Watchpoint on Global Variable *gvar1*.

Flash Memory Programming

Overview

You can program the following in flash:

- Executable or bootable images of applications
- Hardware bitstreams for your FPGA
- File system images, data files such as sample data and algorithmic tables

The executable or bootable images of applications is the most common use case. When the processor in your design comes out of reset, it starts executing code stored in block RAM at the processor reset location. Typically, block RAM size is only a few kilobytes or so and is too small to accommodate your entire software application image. You can store your software application image (typically, a few megabytes-worth of data) in flash memory. A small bootloader is then designed to fit in block RAM. The processor executes the bootloader on reset, which then copies the software application image from flash into external memory. The bootloader then transfers control to the software application to continue execution.

The software application you build from your project is in Executable Linked Format (ELF). When bootloading a software application from flash, ELF images should be converted to one of the common bootloadable image formats, such as Motorola S-record (SREC). This keeps the bootloader smaller and more simple. The Vivado® Design Suite provides interface and command line options for creating bootloaders in SREC format.

Flash Programming from IP Integrator

The IP integrator interface includes dialog boxes from which you can program external Common Flash Interface (CFI) compliant parallel flash devices on your board, connected through the external memory controller (EMC) IP cores. The programming solution is designed to be generic and targets a wide variety of flash hardware and layouts.

The programming is achieved through the debugger connection to a processor in your design. IP integrator downloads and executes a small in-system flash programming stub on the target processor. The in-system programming stub requires a minimum of 8 KB of memory to operate. A host Tcl script drives the in-system flash programming stub with commands and data and completes the flash programming. The flash programming tools do not process or interpret the image file to be programmed, and the tools routinely program the file as-is onto flash memory. Your software and hardware application setup must infer the contents of the file being programmed.

Supported Flash Hardware

The flash programmer uses the Common Flash Interface (CFI) to query the flash devices, so it requires that the flash device be CFI compliant. The layout of the flash devices to form the total memory interface width is also important. The following table lists the supported flash layouts and configurations. If your flash layout does not match a configuration in Table 5-1 you must then customize the flash programming session. Refer to "Customizing Flash Programming" on page 97.

Table 5-1: **Supported Flash Configurations**

x8 only capable device forming an 8-bit data bus
x16/x8 capable device in x8 mode forming an 8-bit data bus
x32/x8 capable device in x8 mode forming an 8-bit data bus
x16/x8 capable device in x16 mode forming a 16-bit data bus
Paired x8 only capable devices forming a 16-bit data bus
Quad x8 only capable devices forming a 32-bit data bus
Paired x16 only capable devices in x16 mode, forming a 32-bit data bus
x32 /x8 capable device in x32 mode, forming a 32-bit data bus
x32 only capable device forming a 32-bit data bus

The physical layout, geometry information, and other logical information, such as command sets, are determined using the CFI. The flash programmer can be used on flash devices that use the CFI-defined command sets only. The CFI-defined command sets are listed in Table 5-2.

Table 5-2: **CFI Defined Command Sets**

CFI Vendor ID	OEM Sponsor	Interface Name
1	Intel/Sharp	Intel/Sharp Extended Command Set
2	AMD/Fujitsu	AMD/Fujitsu Standard Command Set
3	Intel	Intel Standard Command Set
4	AMD/Fujitsu	AMD/Fujitsu Extended Command Set

By default, the flash programmer supports only flash devices which have a sector map that matches what is stored in the CFI table. Some flash vendors have top-boot and bottom-boot flash devices; the same common CFI table is used for both. The field that identifies the boot topology of the current device is not part of the CFI standard. Consequently, the flash programmer encounters issues with such flash devices.

Refer to "Customizing Flash Programming" on page 97 for more information about how to work around the boot topology identification field.

The following assumptions and behaviors apply to programming flash hardware:

- Flash hardware is assumed to be in a reset state when programming is attempted by the flash programming stub.
- Flash sectors are assumed to be in an unprotected state.

The flash programming stub does not attempt to unlock or initialize the flash, and reports an error if the flash hardware is not in a ready and unlocked state.

Note: The flash programmer does not currently support dual-die flash devices which require every flash command to be offset with a Device Base Address (DBA) value. Examples of such dual-die devices are the 512 Mbit density devices in the Intel StrataFlash® Embedded Memory (P30) family of flash memory.

Flash Programmer Performance

The following factors determine the speed at which an image can be programmed:

- The flash programmer communicates with the in-system programming stub using JTAG. Consequently, the inherent bandwidth of the JTAG cable is, in most cases, the bottleneck in programming flash.
- When it is available on the system, it is best to use external memory as scratch memory. This will allow the debugger to download the flash image data without having to stream it in multiple iterations.
- It is desirable to implement the fastest configuration possible when using the MicroBlaze soft processor. You can improve programming speed by turning on features such as the barrel shifter and multiplier.

Customizing Flash Programming

Hardware incompatibilities, flash command set incompatibilities, or memory size constraints are considerations when programming flash. This section briefly describes the flash programming algorithm, so that, if necessary, you can plug in and replace elements of the flow to customize it for your particular setup.

When you click the **Program Flash** button and select a hardware platform project, the following sequence of events occurs:

1. A `flash.tcl` file is written out to the `<hardware platform project>/settings` folder. This contains parameters that describe the flash programming session and is used by the flash programmer Tcl file.
2. IP integrator launches XMD with the flash programmer Tcl script, executing it with a command such as:

   ```
   xmd -nx -hw <hardware platform project>/system.xml -tcl flashwriter.tcl <hardware platform project>/settings/flash.tcl.
   ```

   This flash programmer host Tcl comes from the installation. You can replace the default `flashwriter.tcl` with your own driver Tcl to run when you click the **Program Flash** button by placing a copy of the `flashwriter.tcl` file in your *<hardware platform project>*`/tmp` directory. XMD searches for the specified file in your project directory before looking for it in the installation.

3. The flash programmer Tcl script copies the flash programmer application source files from the installation to the *<hardware platform project>*`/tmp/` folder. It compiles the application locally to execute from the scratch memory address you specified in the dialog box. You can compile your own flash writer sources by modifying your local copy of the `flashwriter.tcl` script to compile your own sources instead of those from the installation.
4. The script downloads the flash programmer to the processor and communicates with the flash programmer through mailboxes in memory.
   In other words, it writes parameters to the memory locations corresponding to variables in the flash programmer address space and lets the flash programmer execute.
5. The script waits for the flash programmer to invoke a callback function at the end of each operation and stops the application at the callback function by setting a breakpoint at the beginning of the function. When the flash programmer stops, the host Tcl processes the results and continues with more commands as required.
6. While running, the flash programmer erases only as many flash blocks as required in which to store the image.

7. The flashwriter allocates a streaming buffer (based on the amount of scratch pad memory available) and iteratively stream programs the image file. The stream buffer is allocated within the flashwriter. If there is enough scratch memory to hold the entire image, the programming can be completed quickly.
8. When the programming is done, the flash programmer Tcl sends an exit command to the flash programmer and terminates the XMD session.

Custom Flow

IP integrator stores the flash settings and temporary files in the hardware platform project directory. If multiple hardware projects exist in the workspace, the flash programmer dialog box prompts you to select the hardware platform. In the following procedure, `<IP integrator project>` refers to this hardware platform project.

1. Create a new subdirectory called `tmp` under the `<IP integrator project>` directory.

   Note: If this folder already exists, skip this step.
2. Copy `flashwriter.tcl` from `<edk_install>/data/xmd/flashwriter.tcl` to your `<IP integrator project>/tmp` directory.
3. Create a `sw_services` directory within your project.
4. Copy the `<edk_install>/data/xmd/flashwriter` directory to the `/sw_services` directory.
5. Change the following line in the `flashwriter.tcl` file copy:

   ```
   set flashwriter_src [file join $xilinx_edk "data" "xmd" "flashwriter" "src"]
   ```

   to

   ```
   set flashwriter_src [file join ".." "sw_services" "flashwriter" "src"]
   ```

From this point when you use the **Program Flash Memory** dialog box in IP integrator , the flash programming tools use the script and the sources you copied into the `sw_services` directory. You can customize these as required.

If you prefer to not have the GUI overwrite the `<IP integrator project>/settings/flash.tcl` file and only use the values that you specified in the `flash.tcl` file, on the command line, run the command:

```
xmd -nx -hw system.xml -tcl tmp/flashwriter.tcl settings/flash.tcl
```

Table 5-3 lists the available parameters in the <IP integrator project> directory.

Table 5-3: **Flash Programming Parameters**

Variable	Function
EXTRA_COMPILER_FLAGS	For MicroBlaze, specify any compiler flags required to turn on support for hardware features. For example, if you have the hardware multiplier enabled, add -mno-xl-soft-mul here.
FLASH_BASEADDR	The base address of the flash memory bank.
FLASH_BOOT_CONFIG	Refer to "Handling Flash Devices with Conflicting Sector Layouts" on page 100.
FLASH_FILE	A string containing the full path of the file to be programmed.
FLASH_PROG_OFFSET	The offset within the flash memory bank at which the programming should be done.
PROC_INSTANCE	The instance name of the processor used for programming.
SCRATCH_BASEADDR	The base address of the scratch memory used during programming.
SCRATCH_LEN	The length of the scratch memory in bytes.
TARGET_TYPE	The type of the processor instance used for programming: MicroBlaze or Cortex A9 processor.
XILINX_PLATFORM_FLASH	To enable use of the Xilinx Platform Flash XL flash device.
XMD_CONNECT	The connect command used in XMD to connect to the processor.

Manual Conversion of ELF Files to SREC for Bootloader Applications

If you want to create SREC images of your ELF file manually instead of using the auto-convert feature in IP integrator you can use the command line tools. For example, to create a final software application image named myexecutable.elf, navigate in the console of your operating system (Cygwin on Windows platforms) to the folder containing this ELF file and type the following:

```
<platform>-objcopy -O srec myexecutable.elf myexecutable.srec
```

where <platform> is **mb** if your processor is a MicroBlaze.

This creates an SREC file that you can then use as appropriate. The mb-objcopy utility is a GNU binary that ships with Vivado.

Operational Characteristics and Workarounds

Handling Xilinx Platform Flash Modes

Xilinx Platform Flash memory devices initialize in synchronous mode. You must set these devices to asynchronous mode before performing device operations. When using the Xilinx Software Development Kit, you can select a check box to inform the Flash programming interface to treat the target device as Xilinx Platform Flash. This setting enables an internal workaround in the programmer that sets the device to asynchronous mode before programming.

Handling Flash Devices with 0xF0 as the Read-Reset Command

The CFI specification defines the read-reset command as `0xFF` / `0xF0`. By default the flash programmer uses the `0xFF` read-reset command. Certain devices require `0xF0` as the read-reset command, however, the flash programmer is unable to determine this automatically. Consequently, you might encounter issues when programming newer devices.

In that event of an error occurring follow the documented steps in Customizing Flash Programming, page 97, then modify the `#define FRR_CMD 0xFF` in the `cfi.c` file to `#define FRR_CMD 0xF0`.

Handling Flash Devices with Conflicting Sector Layouts

Some flash vendors store a different sector map in the CFI table and another (based on the boot topology of the flash device) in hardware. Because the boot topology information is not standardized in CFI, the flash programmer cannot determine the layout of your particular flash device.

If your flash hardware has a sector layout that is different from the one specified in the CFI table for the device, then you must create a custom flash programming flow. You must determine whether the device is a top-boot or a bottom-boot flash device.

In a top-boot flash device, the smallest sectors are the last sectors in the flash. In a bottom-boot flash device, the smallest sectors are the first sectors in the flash layout.

After you determine the flash device type, you must copy over the files to create a custom programming flow.

- If you have a bottom-boot flash, add the following line in your `/etc/flash_params.tcl` file:

  ```
  set FLASH_BOOT_CONFIG BOTTOM_BOOT_FLASH
  ```

- If you have a top-boot flash, add the following line in your `/etc/flash_params.tcl` file:

  ```
  set FLASH_BOOT_CONFIG TOP_BOOT_FLASH
  ```

Next, run the flash programming from the command line with the following command:

```
xmd -tcl flashwriter.tcl
```

Internally, these variables cause the flash programmer to rearrange the sector map according to the boot topology.

Data Polling Algorithm for AMD/Fujitsu Command Set

The DQ7 data polling algorithm is used during erasure and programming operations on flash hardware that supports the AMD/Fujitsu command set.

Certain flash devices are known to use a configuration register to control the behavior of the data polling DQ7 bit. Some known flash devices that offer this configuration register feature are: AT49BV322A(T), AT49BV162A(T), and AT49BV163A(T).

It is required that DQ7 output `0` during an erase operation and `1` at the end of the operation. Similarly, DQ7 must output inverted data during programming and the actual data after programming is done. If your flash hardware has a different configuration when using the Program Flash Memory dialog box, then the programming could fail.

Refer to your flash hardware datasheet for information about how to reset the configuration so that DQ7 has the appropriate outputs upon erasure and ending.

GNU Utilities

This appendix describes the GNU utilities available for use with the Vivado® Design Suite.

General Purpose Utility for MicroBlaze Processors

cpp

Pre-processor for C and C++ utilities. The preprocessor is invoked automatically by GNU Compiler Collection (GCC) and implements directives such as file-include and define.

gcov

This is a program used in conjunction with GCC to profile and analyze test coverage of programs. It can also be used with the `gprof` profiling program.

Note: The `gcov` utility is not supported by IP integrator or SDK, but is provided as is for use if you want to roll your own coverage flows.

Utilities Specific to MicroBlaze Processors

Utilities specific to MicroBlaze™ Processors have the prefix "`mb-`," as shown in the following program names.

mb-addr2line

This program uses debugging information in the executable to translate a program address into a corresponding line number and file name.

mb-ar

This program creates, modifies, and extracts files from archives. An archive is a file that contains one or more other files, typically object files for libraries.

mb-as

This is the assembler program.

mb-c++

This is the same cross compiler as `mb-gcc`, invoked with the programming language set to C++. This is the same as `mb-g++`.

mb-c++filt

This program performs name demangling for C++ and Java function names in assembly listings.

mb-g++

This is the same cross compiler as `mb-gcc`, invoked with the programming language set to C++. This is the same as `mb-c++`.

mb-gasp

This is the macro preprocessor for the assembler program.

mb-gcc

This is the cross compiler for C and C++ programs. It automatically identifies the programming language used based on the file extension.

mb-gdb

This is the debugger for programs.

mb-gprof

This is a profiling program that allows you to analyze how much time is spent in each part of your program. It is useful for optimizing run time.

mb-ld

This is the linker program. It combines library and object files, performing any relocation necessary, and generates an executable file.

mb-nm

This program lists the symbols in an object file.

mb-objcopy

This program translates the contents of an object file from one format to another.

mb-objdump

This program displays information about an object file. This is very useful in debugging programs, and is typically used to verify that the correct utilities and data are in the correct memory location.

mb-ranlib

This program creates an index for an archive file, and adds this index to the archive file itself. This allows the linker to speed up the process of linking to the library represented by the archive.

mb-readelf

This program displays information about an Executable Linked Format (ELF) file.

mb-size

This program lists the size of each section in the object file. This is useful to determine the static memory requirements for utilities and data.

mb-strings

This is a useful program for determining the contents of binary files. It lists the strings of printable characters in an object file.

mb-strip

This program removes all symbols from object files. It can be used to reduce the size of the file, and to prevent others from viewing the symbolic information in the file.

Other Programs and Files

The following Tcl and Tk shells are invoked by various front-end programs:

- `cygitclsh30`
- `cygitkwish30`
- `cygtclsh80`
- `cygwish80`
- `tix4180`

Appendix B

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

The following Vivado® Design Suite guides are referenced in this document.

1. *Software Development Kit (SDK) Help* (UG782)
2. *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585)
3. *MicroBlaze Processor User Guide* (UG081)

Other Xilinx Documentation

- *Vivado Design Suite User Guide: Embedded Hardware Design* (UG898)
- *Vivado Design Suite Tutorial: Embedded Hardware Design* (UG940)
- *Generating Basic Software Platforms Reference Guide* (UG1138)
- *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821)

Additional Resources

- GNU website: http://www.gnu.org

- Red Hat Insight website: http://sources.redhat.com/insight.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

© Copyright 2014 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.